

MANNING

WINDOWS POWERSHELL

实战指南

(第2版)

Learn WINDOWS **POWERSHELL** IN A MONTH OF LUNCHES *Second Edition*

[美] Don Jones Jeffery Hicks 著

宋沅剑 何文通 黄钊吉 陈畅亮 译



中国工信出版集团



人民邮电出版社
POSTS & TELECOM PRESS

目 录

[版权信息](#)

[版权声明](#)

[内容提要](#)

[第一版赞誉](#)

[前言](#)

[关于本书](#)

[作者在线](#)

[关于作者](#)

[关于译者](#)

[鸣谢](#)

[第1章 背景介绍](#)

[1.1 为什么要重视PowerShell](#)

[1.2 本书适用读者](#)

[1.3 如何使用本书](#)

[1.4 搭建自己的实验环境](#)

[1.5 安装Windows PowerShell](#)

[1.6 在线资源](#)

[1.7 赶紧使用PowerShell吧](#)

[第2章 初识PowerShell](#)

[2.1 选择你的“武器”](#)

[2.2 重新认识代码输入](#)

[2.3 常见误区](#)

[2.4 如何查看当前版本](#)

[2.5 动手实验](#)

[2.6 进一步学习](#)

[第3章 使用帮助系统](#)

[3.1 帮助系统：发现命令的方法](#)

[3.2 可更新的帮助](#)

[3.3 查看帮助](#)

[3.4 使用帮助找命令](#)

[3.5 详解帮助](#)

[3.6 访问“关于”主题](#)

[3.7 访问在线帮助](#)

[3.8 动手实验](#)

[第4章 运行命令](#)

[4.1 无需脚本，仅仅是运行命令](#)

[4.2 剖析一个命令](#)

[4.3 Cmdlet命名惯例](#)

[4.4 别名：命令的昵称](#)

[4.5 使用快捷方式](#)

[4.6 小小作弊一下：Show-Command](#)

[4.7 对扩展命令的支持](#)

[4.8 处理错误](#)

[4.9 常见误区](#)

[4.10 动手实验](#)

[第5章 使用提供程序](#)

[5.1 什么是提供程序](#)

[5.2 FileSystem的结构](#)

[5.3 文件系统——其他数据存储的模板](#)

[5.4 使用文件系统](#)

[5.5 使用通配符以及绝对路径](#)

[5.6 使用其他提供程序](#)

[5.7 动手实验](#)

[5.8 进一步学习](#)

[第6章 管道：连接命令](#)

[6.1 一个命令与另外一个命令连接：为你减负](#)

[6.2 输出结果到CSV或XML文件](#)

[6.3 管道传输到文件或打印机](#)

[6.4 转换成HTML](#)

[6.5 使用Cmdlets修改系统：终止进程和停止服务](#)

[6.6 常见误区](#)

[6.7 动手实验](#)

[第7章 扩展命令](#)

[7.1 如何让一个Shell完成所有事情](#)

[7.2 关于产品的“管理Shell”](#)

[7.3 扩展：找到并添加插件](#)

[7.4 扩展：找到并添加模块](#)

[7.5 命令冲突和移除扩展](#)

[7.6 玩转一个新的模块](#)

[7.7 配置脚本：在启动Shell时预加载扩展](#)

[7.8 常见误区](#)

[7.9 动手实验](#)

[第8章 对象：数据的另一个名称](#)

[8.1 什么是对象](#)

[8.2 为什么PowerShell使用对象](#)

[8.3 探索对象：Get-Member](#)

[8.4 对象标签，也就是所谓的“属性”](#)

[8.5 对象行为，也就是所谓的“方法”](#)

[8.6 排序对象](#)

[8.7 选择所需的属性](#)

[8.8 在命令结束之前总是对象的形式](#)

[8.9 常见误区](#)

[8.10 动手实验](#)

[第9章 深入理解管道](#)

[9.1 管道：更少的输入，更强大的功能](#)

[9.2 PowerShell如何传输数据给管道](#)

[9.3 方案A：使用ByValue进行管道输入](#)

[9.4 方案B：使用ByPropertyName进行管道传输](#)

[9.5 数据不对齐时：自定义属性](#)

[9.6 括号命令](#)

[9.7 提取属性的值](#)

[9.8 动手实验](#)

[9.9 进一步学习](#)

[第10章 格式化及如何正确使用](#)

[10.1 格式化：让输出更加美观](#)

[10.2 默认格式](#)

[10.3 格式化表格](#)

[10.4 格式化列表](#)

[10.5 宽度的格式化](#)

[10.6 定制列和列表记录](#)

[10.7 输出到文件、打印机或者主机上](#)

[10.8 另外一个输出：网格](#)

[10.9 常见误区](#)

[10.10 动手实验](#)

[10.11 进一步学习](#)

[第11章 过滤和对比](#)

[11.1 只获取必要的内容](#)

- [11.2 左过滤](#)
- [11.3 对比操作符](#)
- [11.4 过滤对象的管道](#)
- [11.5 迭代的命令行模式](#)
- [11.6 常见误区](#)
- [11.7 动手实验](#)
- [11.8 进一步学习](#)

[第12章 学以致用](#)

- [12.1 定义任务](#)
- [12.2 发现命令](#)
- [12.3 学习如何使用命令](#)
- [12.4 自学的一些技巧](#)
- [12.5 动手实验](#)

[第13章 远程处理：一对一及一对多](#)

- [13.1 PowerShell远程处理的原理](#)
- [13.2 WinRM概述](#)
- [13.3 一对一场景的Enter-PSSession和Exit-PSSession](#)
- [13.4 一对多场景的Invoke-Command](#)
- [13.5 远程命令和本地命令之间的差异](#)
- [13.6 深入探讨](#)
- [13.7 远程处理的配置选项](#)
- [13.8 常见误区](#)
- [13.9 动手实验](#)
- [13.10 进一步学习](#)

[第14章 Windows管理规范](#)

- [14.1 WMI概要](#)
- [14.2 关于WMI的坏消息](#)
- [14.3 探索WMI](#)
- [14.4 选择你的武器：WMI或CIM](#)
- [14.5 使用Get-WmiObject](#)
- [14.6 使用Get-CimInstance](#)
- [14.7 WMI文档](#)
- [14.8 常见误区](#)
- [14.9 动手实验](#)
- [14.10 进一步学习](#)

[第15章 多任务后台作业](#)

- [15.1 利用PowerShell实现多任务同时处理](#)

- [15.2 同步VS异步](#)
- [15.3 创建本地作业](#)
- [15.4 WMI作业](#)
- [15.5 远程处理作业](#)
- [15.6 获取作业执行结果](#)
- [15.7 使用子作业](#)
- [15.8 管理作业的命令](#)
- [15.9 调度作业](#)
- [15.10 常见困惑点](#)
- [15.11 动手实验](#)

[第16章 同时处理多个对象](#)

- [16.1 对于大量管理的自动化](#)
- [16.2 首选方法：“批处理”Cmdlet](#)
- [16.3 MI方式：调用WMI方法](#)
- [16.4 后备计划：枚举对象](#)
- [16.5 常见误区](#)
- [16.6 动手实验](#)

[第17章 安全警报](#)

- [17.1 保证Shell安全](#)
- [17.2 Windows PowerShell的安全目标](#)
- [17.3 执行策略和代码签名](#)
- [17.4 其他安全措施](#)
- [17.5 其他安全漏洞](#)
- [17.6 安全建议](#)
- [17.7 动手实验](#)

[第18章 变量：一个存放资料的地方](#)

- [18.1 变量简介](#)
- [18.2 存储值到变量中](#)
- [18.3 使用变量：有趣的引号](#)
- [18.4 存储多个对象在一个变量中](#)
- [18.5 双引号的其他技巧](#)
- [18.6 声明变量类型](#)
- [18.7 与变量相关的命令](#)
- [18.8 针对变量的最佳实践](#)
- [18.9 常见误区](#)
- [18.10 动手实验](#)
- [18.11 进一步学习](#)

第19章 输入和输出

19.1 提示并显示信息

19.2 Read-Host命令

19.3 Write-Host命令

19.4 Write-Output命令

19.5 其他写入的方式

19.6 动手实验

19.7 进一步学习

第20章 轻松实现远程控制

20.1 PoweShell远程控制稍微容易一点

20.2 创建并使用可重用会话

20.3 利用Enter-PSSession命令使用会话

20.4 利用Invoke-Command命令使用会话

20.5 隐式远程控制：导入一个会话

20.6 断开会话

20.7 动手实验

20.8 进一步学习

第21章 你把这叫作脚本

21.1 非编程，而更像是批处理文件

21.2 使得命令可重复执行

21.3 参数化命令

21.4 创建一个带参数的脚本

21.5 为脚本添加文档

21.6 一个脚本，一个管道

21.7 作用域初探

21.8 动手实验

第22章 优化可传参脚本

22.1 起点

22.2 让PowerShell去做最难的工作

22.3 将参数定义为强制化参数

22.4 添加参数别名

22.5 验证输入的参数

22.6 通过添加详细输出获得用户友好体验

22.7 动手实验

第23章 高级远程配置

23.1 使用其他端点

23.2 创建自定义端点

[23.3 启用多跳远程 \(multi-hop remoting\)](#)

[23.4 深入远程身份验证](#)

[23.5 动手实验](#)

[第24章 使用正则表达式解析文本文](#)

[24.1 正则表达式的目标](#)

[24.2 正则表达式入门](#)

[24.3 通过-Match使用正则表达式](#)

[24.4 通过Select-String使用正则表达式](#)

[24.5 动手实验](#)

[24.6 进一步学习](#)

[第25章 额外的提示、技巧以及技术](#)

[25.1 Profile、提示以及颜色：自定义Shell界面](#)

[25.2 运算符：-AS,-IS,-Replace,-Join,-Split,-IN,-Contains](#)

[25.3 字符串处理](#)

[25.4 日期处理](#)

[25.5 处理WMI日期](#)

[25.6 设置参数默认值](#)

[25.7 学习脚本块](#)

[25.8 更多的提示、技巧及技术](#)

[第26章 使用他人的脚本](#)

[26.1 脚本](#)

[26.2 逐行检查](#)

[26.3 动手实验](#)

[第27章 学无止境](#)

[27.1 进一步学习的思想](#)

[27.2 既然已经阅读了本书，那么我要从哪里开始呢](#)

[27.3 你会喜欢的其他资源](#)

[第28章 PowerShell备忘清单](#)

[28.1 标点符号](#)

[28.2 帮助文档](#)

[28.3 运算符](#)

[28.4 自定义属性与列的语法](#)

[28.5 管道参数输入](#)

[28.6 何时使用\\$](#)

[附录 复习实验](#)

[实验回顾1：第1—6章](#)

[实验回顾2：第1—14章](#)

实验回顾3: 第1—19章
看完了

版权信息

书名：Windows PowerShell实战指南（第2版）

ISBN：978-7-115-40967-6

本书由人民邮电出版社发行数字版。版权所有，侵权必究。

您购买的人民邮电出版社电子书仅供您个人使用，未经授权，不得以任何方式复制和传播本书内容。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。

• 著 [美] Don Jones Jeffery Hicks

译 宋沅剑 何文通 黄钊吉 陈畅亮

责任编辑 王峰松

• 人民邮电出版社出版发行 北京市丰台区成寿寺路11号

邮编 100164 电子邮件 315@ptpress.com.cn

网址 <http://www.ptpress.com.cn>

• 读者服务热线：(010)81055410

反盗版热线：(010)81055315

版权声明

Original English language edition, entitled Learn Windows PowerShell in a Month of Lunches, 2nd Edition by Don Jones & Jeffery Hicks, published by Manning Publications, USA. Copyright © 2014 by Manning Publications. Simplified Chinese-language edition, Copyright © 2015 by Posts & Telecom Press. All rights reserved.

本书中文简体字版由Manning Publications授权人民邮电出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

版权所有，侵权必究。

内容提要

PowerShell既是编程语言，也是一种管理Shell。通过PowerShell几乎可以管理Windows的方方面面。本书是为忙于运维的管理人员所编写的参考指南。只需要1个月、每天1小时，读者就能够学到让自己的工作变得更轻松的实战技能。本书章节安排合理，每章只需要1小时，即可以零编程基础开始学习PowerShell。本书作者是PowerShell界的泰斗Don Jones与Jeffery Hicks。他们都是多年的PowerShell MVP，并以简洁、易入门的培训和写作风格而著称。

第一版赞誉

本书的内容在我看来就像在教室上课一样生动，通过本书包含的大量实践练习，由浅入深地体会PowerShell的力量。

——Chuck Durfee

Graebel公司，高级软件工程师

由新手到高手的过程中，本书是你唯一需要的。通过阅读本书，你就能够知道Don Jones为什么是一名PowerShell界的明星。

——David Moravec

独立博客PowerShell.cz，SCCM管理员

学习PowerShell的核心指南！强烈推荐！

——Ray Booysen

法国巴黎银行BNP Paribas，开发工程师

真希望我在开始学习PowerShell时就能够看到这本书。

——Richard Siddaway

微软PowerShell MVP，IT架构师

本书不仅教会你PowerShell，还教会你如何成为PowerShell的专家。

——Nikander Bruggeman和Margriet Bruggeman

Lois & Clark IT服务公司，.NET咨询顾问

前言

我们已经从事PowerShell教学和写作很长时间。当Don开始规划本书的第一版时，他意识到大多数PowerShell作者和讲师——包括他自己——会强迫学生将Shell作为一门编程语言学习。大多数PowerShell书籍都会通过三章或者四章进入“脚本”主题，而现在越来越多的PowerShell学习者对面向编程的学习方法避之不及。这些学生只是想将Shell作为Shell使用，至少在一开始是这样的。我们只是希望提供符合该要求的学习体验。

所以Don希望尝试这种方法。通过在WindowsITPro.com发布本书的目录，来自博客读者的大量反馈最终让本书变得更好。他希望每一章短小、目的明确且短时间内就可以掌握——他知道管理员们并没有多少闲暇时间，通常他们都是在需要的时候才会去学习。当PowerShell v3发布后，这明显是更新本书的最好时机，Don最终找到他的长期合作伙伴Jeffery Hicks共同完成本书。

我们希望本书专注于PowerShell本身，而不是大量PowerShell可以用到的技术上，比如Exchange Server、SQL Server、System Center等。我们真心认为通过学会正确使用Shell，你就可以通过自学掌握所有这些可以通过PowerShell使用的服务器级别产品。所以本书重点是使用PowerShell所需的核心技能。即使你还使用了“cookbook”风格的书籍（该类书中为特定管理任务提供了直接可以上手使用的答案），本书也可以帮助你理解那些书中实例的原理。对例子的理解能够帮助你更容易修改这些示例，从而完成其他任务，最终你可以从无到有构建你自己的命令。

我们希望本书不是你学习PowerShell的唯一工具。实际上，在本书提供的网站上（也就是MoreLunches.com）还提供了大量简短的内容帮助你更好地学习PowerShell。该网站提供了与本书章节对应的免费视频，从中你可以看到和听到对于核心技术的实践。我们还共同编著了《Learn PowerShell Toolmaking in a Month of Lunches》。该书同样以一天一次的方式提供了学习PowerShell脚本以及工具制作的能力。

如果你还需要其他额外帮助，我们希望你登录 [\[www.PowerShell.org\]](http://www.PowerShell.org) (www.PowerShell.org)。我们在该网站的多个讨论组回答问题。我们会非常高兴在你遇到任何问题卡住时拯救你。该网站还是强大活跃的PowerShell社区入口——你可以学习关于年度脚本游戏，也就是线下的PowerShell峰会，以及所有关于各个区域及本地用户组举行的PowerShell相关的活动。请加入——这是将PowerShell作为你职业生涯更强大的组成部分的方法。

请享受本书——在学习使用Shell的过程中祝你好运。

关于本书

关于本书中大多数你所需知道的内容都在第1章中进行描述，但有一些事需要提前告知。

首先，如果你计划跟随我们的示例并完成动手实验，你需要一台运行**Windows 8**或**Windows Server 2012**的计算机或虚拟机。我们在第1章中进行了更详细的阐述。你也可以在**Windows 7**上运行这些示例，但在动手实验中有一些知识点无法进行实验。当然，使用更新版本的操作系统也是可以的，比如**Windows 8.1**或**Windows Server 2012 R2**。本书涵盖了**Windows PowerShell**第三版以及更新的版本；后续版本仅仅是添加了新的功能，因此本书同样适用。

其次，请准备好从头到尾，按照章节先后顺序阅读本书。同样，我们在第一章中会进行详细解释，但背后的思想是每一章都会介绍一些新的内容，这些内容都会在下一章中被用到。请不要尝试一次性阅读完整本书——请坚持每天一章的方式。人的大脑一次只能理解有限的信息，通过将**PowerShell**分解为小的片段，你实际上可以更快、更彻底地学习**PowerShell**。

再次，本书包含大量的代码段。大多数代码段较短，因此你可以很容易地输入这些代码。实际上，我们推荐你手工敲一遍代码，这样做可以巩固核心**PowerShell**技能：准确地输入！较长的代码段也同样在代码清单中且可以从<http://Morelunches.com>（只需通过单击本书的封面图片并找到“下载”部分）进行下载，也可以通过出版社的网站 www.manning.com/LearnWindowsPowerShellinaMonthofLunchesSecondEdition 进行下载。

也就是说，还有一些需要注意的惯例。代码总是以特殊字体进行显示，例子如下：

```
Get-WmiObject -class Win32_OperatingSystem  
➡ -computerName SERVER-R2
```

本示例还描述了在本书中使用的行继续符。这意味着这两行在PowerShell中实际上是作为一行进行输入。换句话说，不要在Win32_OperationSystem后敲击回车键或返回键——而是在该语句右侧继续进行输入。PowerShell允许行非常长，但本书的纸张却不能容纳那么长。

有时，你还能在本书中看到代码字体，如当我们写Get-Command时。这只是为了让你知道你正在查看的是一个命令、参数或其他你将会在Shell中输入的元素。

然后是一个我们在很多章节使用的有点让人难以琢磨的主题：重音符（`）。下面是示例：

```
Invoke-Command -scriptblock { Dir } `
-computerName SERVER-R2,localhost
```

该字符在第一行的最后并不是洒出来的墨水——而是你需要输入的实际符号。在美式键盘中，重音符（或者称为沉音符）通常位于键盘的左上部分，在Esc键下面，和波浪号（~）位于同一个键位。当你在代码清单中看到重音符时，请按照原样输入它。此外，当该字符出现于行尾时——正如之前示例所示——请确保该字符是行的最后一个字符。如果在该字符之后又存在任何空格或Tab符号，重音符则无法正常生效。在本书代码段的重音符之后不会存在空格或者Tab符号。

最后，我们将会偶尔将你导向到Internet资源上。这些URL会很长并难以输入。我们会将这些URL替换为基于Manning出版社的短链接，看上去就像<http://mng.bz/S085>（你会在第1章中看到该链接）。

作者在线

购买Learn Windows PowerShell in a Month of Lunches（Second Edition）还包含了访问由Manning出版社运营的私有论坛。在该论坛中，你可以对本书进行评价、提出技术问题并得到作者和其他用户的帮助。通过www.manning.com/LearnWindowsPowerShellinaMonthofLunchesSecondE

dition或www.manning.com/jones3并单击Author Online链接来访问和订阅论坛。该页面提供了在注册后如何访问论坛的信息，以及可以得到的帮助的类型与论坛行为规范。

Manning对读者的承诺是提供一个交流的场所。在该场所，读者和读者以及读者和作者之间可以进行有价值的对话。但并不承诺有多少代表作者的参与者参与论坛，作者参与论坛都是志愿的（且不收报酬）。我们建议你尝试问作者一些有挑战性的问题，从而使他们保持兴趣。

作者在线论坛以及之前讨论内容的存档，在本书印刷时，就可以通过出版社的网站进行访问。

关于作者

本书作者是PowerShell界的泰斗Don Jones与Jeffery Hicks，他们俩都是多年的PowerShell MVP，并以简洁、易入门的培训和写作风格而著称。Don在PowerShell.org撰写博客，而Jeff的博客则是 jdhitsolutions.com/blog。

由于Don Jones在Windows PowerShell方面的工作，他多年获得微软公司最有价值专家（MVP）奖项。他是微软TechNet杂志的Windows PowerShell专栏的作者，在PowerShell.org写博客。他创作出版了“Decision Maker”专栏，并为Redmond杂志写博客。Don是一名多产的技术作者，自2001年以来出版了超过12本书。他还是Concentrated Technology (ConcentratedTech.com)公司的首席技术专家和高级合伙人。该公司是一家IT教育和战略咨询公司。Don使用的第一个Windows脚本语言是KiXtart，该语言可追溯至20世纪90年代中期。很快他就在1995年转移使用VBScript。他还是最早期使用微软代码名称为“Monad”产品的IT专家之一——该产品后来成为Windows PowerShell。Don住在拉斯维加斯，在世界各地提供IT培训（尤其是PowerShell方面），并在IT峰会上进行演讲。

Jeffery Hicks在Windows PowerShell方面连续多年获得微软最有价值专家奖项。他还是一个微软认证讲师以及拥有20年经验的IT老兵，大多数工作经验花在微软服务器技术的咨询上。现在他作为独立作者、培训师顾问，为全世界的客户提供服务。Jeffery为MPCMag.com中流行的Prof.PowerShell专栏撰写文章，并且是Petri IT知识库的常规贡献者。如果他不在写书，更可能是他正在为诸如TrainSignal之类的公司录制培训视频或在讨论论坛中帮助他人。你可以在Jeffery的博客 <http://jdhitsolutions.com/blog> 中查看他的最新状态。

关于译者

宋沅剑，微软SQL Server MVP，SQL Server 2012 UI审查专家，Professional Association for SQL Server(PASS)北京分会联合发起人，数据库大会、TechED特约讲师，目前就职于易车网，负责易车海量数据平台的运维工作，设计自动化监控运维方案。曾任数据库高级顾问，帮助国内超过50家客户设计高可用/灾备方案。

何文通，新蛋科技有限公司信息系统部数据库管理科科长，曾任职于大型制造业公司，负责数据库运维与管理。加入新蛋公司后，主要负责MS SQL数据库运维工作，在MS SQL性能优化与故障诊断方面有丰富的实战经验。

黄钊吉，近8年数据库相关经验，目前任职于一家大型物流企业，负责数据平台规划和运维工作。主要研究SQL Server性能和高可用技术，是三届SQL Server MVP，CSDN论坛SQL版大版版主，CSDN博客专家，《SQL Server性能优化与管理的艺术》一书作者。

陈畅亮，SQL Server MVP，曾受邀参加2015年DTCC（中国数据库技术大会）作为演讲嘉宾，出版书籍《SQL Server性能调优实战》，主要研究SQL Server、MySQL、NoSQL，以及分布式环境下海量数据存储的设计与开发。

鸣谢

书当然不会自行书写、编辑和出版。Don希望感谢在Manning出版社那些决定在PowerShell不同种类书籍都碰碰运气的所有人，以及那些努力帮助完成本书的人。Jeffery希望感谢Don邀请他参与完成本书，并感谢所有的PowerShell社区的激情与支持。Don和Jeffery都对Manning出版社让他们继续“一个月的午餐”系列第二版心怀感激。

也感谢所有在书写阶段阅读手稿并参与审阅的同僚：Bennett Scharf, Dave Pawson, David Moravec, Keith Hill, and Rajesh Attaluri; 还感谢James Berkenbile 和Trent Whiteley对手稿和代码的技术审阅。

第1章 背景介绍

自从2006年第一版Windows PowerShell面世以来，我们就一直在致力于对该技术进行教学推广。那时候，PowerShell的大部分使用者都是长期使用VBScript的用户，而且他们也非常期待能通过对VBScript的熟悉来学习PowerShell。于是，开展培训以及编写PowerShell书籍的作者都采用了一种和其他编程语言教学一样的方式来教学PowerShell。

但是从2009年开始发生了一些改变。越来越多没有VBScript经验的人开始学习PowerShell这门语言。因为之前我们主要关注于脚本的编写，所以对PowerShell的教学不再那么卓有成效。也就是在那个时候，我们意识到PowerShell并不仅仅是一门脚本语言，其实是一种运行命令行工具的命令行Shell。和其他优秀的Shell一样，虽然PowerShell可以通过脚本实现很复杂的功能，但脚本仅是使用PowerShell的一种方式，因此学习PowerShell并不一定需要从脚本开始。之后，我们在每年的技术演讲会议上逐渐改变了我们的教学方式，同时也将这些教学方式的变化体现在我们的教学课程中。最后，我们出版了这本书，这也是我们想出的针对非编程背景的人员教学PowerShell的最好方式。但是在开始学习之前，我们需要了解一下背景。

1.1 为什么要重视PowerShell

从Batch、KiXtart、VBScript到现在，可以看到Windows PowerShell并不是微软（或者其他公司）首次为Windows管理员提供自动化管理的工具。我们认为，有必要让你们了解为什么需要关注PowerShell这个工具。因为当你们这样做的时候，会发现花费一定的时间去学习PowerShell是值得的。想象一下，在没有使用PowerShell之前我们的工作是怎样的，在使用该工具后又有哪些变化。

没有PowerShell

Windows操作系统管理员总是喜欢通过单击用户图形化界面去完成他们的工作。GUI（用户图形化界面）是Windows操作系统的一个

最大的特点——毕竟这个操作系统并不是“文字模式”。因为GUI总是让我们很轻易找到我们能做的一切，所以它是那么强大。笔者仍然还记得第一次展开活动目录下的用户和计算机的场景。通过单击各种按钮，阅读工具栏提示信息，选择下拉菜单，右键单击某些图标，来查看用户与计算机中的各项功能。GUI是使得我们能够更容易学习的一种工具。但是不幸的是，GUI并不能带来任何效率提升上的回报。如你花费5分钟在活动目录中创建一个新的用户（合理的设想下，需要填写大量的信息），之后再新建用户时，也不会更快。那么新建100个新用户就会花费500分钟来完成——没有其他任何办法使得我们输入信息以及单击操作更快，从而加快该过程。

微软之前也尝试去解决该问题，VBScript可能算是其中最成功的一次尝试。如果你需要花费一小时去编写一条VBScript语句来将CSV文件中的新用户导入到活动目录中，但是以后你可能只需要花费几秒钟就可以完成同样的工作。VBScript的问题在于微软没有全心全意地对其提供支持，微软需要确保各种对象都可以通过VBScript访问、调用，而如果开发人员因为时间的原因或者是忘记这块知识，那么你就只能卡在那儿了。例如，想通过VBScript修改网卡IP，没问题。但是，想检查网络连接的速度，那就不行了，因为没人记得可以把这个功能设置为VBScript可访问的形式。这也算是一种遗憾。Jeffery Snover，Windows PowerShell的架构师称之为“最后一英里”。你可以通过VBScript（或者其他类似的技术）来做很多事情，但是在某些时刻总会让人失望，从来不会让我们顺利通过“最后一英里”完成之后的工作。

Windows PowerShell正是微软公司试图改善这一缺陷的尝试，让你顺利通过“最后一英里”，进而完成工作。

拥有PowerShell

微软对Windows PowerShell的定位是我们可以通过该Shell完全管理Windows系统中的功能。微软仍在继续开发GUI的控制台，但是底层执行的仍然是PowerShell命令。通过这种方式，微软保证我们可以在该Shell中完成Windows系统中任意的工作。如果需要自动化一个重复性的任务或者完成在GUI中不支持的工作，那么你可以使用该Shell来达到所愿。

很多微软的产品都已经采用了这种开发方法，如Exchange Server 2007和2010、Sharepoint Server 2010、大部分System Center产品以及Windows系统中大量的组件。接下来，越来越多的产品和Windows系统中组件会采用这个Shell。Windows Server 2012（首次采用PowerShell V3）甚至可以完全通过PowerShell或者使用基于PowerShell的GUI工具来进行管理。这也就是为什么我们要重视PowerShell。在接下来的几年，PowerShell会成为越来越多的管理功能的底层实现。

此时，我们仔细想想：如果你正在管理一个拥有很多IT工程师的团队，你希望谁的职级更高，希望谁能拿更多的薪水，是每次都要花费几分钟使用GUI来完成一个任务的人，还是一个可以通过脚本花费几秒钟自动化完成的人？无论你是来自哪个领域的IT从业人员，我们都知道应该如何选择。询问一个思科的管理员、AS/400的操作员或者Unix管理员，他们都会回答“我更希望选择可以借助命令行更有效率地完成工作的人员”。以后的Windows系统工程师可以简单分为两类，一部分会使用PowerShell，另一部分则不会。正如Don在微软2010TechEd会议上著名的言论：我们的选择是“学习PowerShell”，还是“来包炸薯条”？

我们很欣慰，你已经决定来学习PowerShell。

1.2 本书适用读者

这本书并不是适合所有人。实际上，微软PowerShell团队已经定义了三类适用PowerShell的人群：

- 主要使用命令行以及采用第三方开发的工具的管理员；
- 能将命令行和工具集成为一个更复杂的工具（之后那些缺乏经验的成员可以立即使用这个工具来完成相关工作）的管理员；
- 开发可重复使用的工具或者程序的管理员或者开发人员。

本书主要是针对第一类人编写的。包括开发者在内的所有人去理解该Shell如何执行命令是非常有必要的。毕竟，如果你正准备去开发一个工具或者编写一些命令，那么你应该知道这个Shell是如何运行的，这样可以保证开发出来的工具或者命令能像在Shell中运行得那么顺畅。

如果你对如何利用命令行来完成复杂的命令感兴趣，比如新建一个用户，在学习完本书后，你可以看到如何实现该功能。甚至你可以编写自己的脚本，并且该脚本可以让其他管理员任意使用。但是本书并不会很深入地讲解PowerShell的每项功能。我们的宗旨是让你能够使用该Shell，并能立即应用到生产环境。

我们也会使用多种方法来演示如何将PowerShell关联到其他的管理工具。在后续章节中，我们会以WMI（Windows Management Instrumentation）以及常用的命令作为示例。大体上，我们仅会介绍PowerShell可以与哪些技术进行关联，并且讲解它们之间是如何进行关联的。其实，这些主题甚至都可以单独出书介绍（我们会在本书适当的地方给出对应的建议）。在本书中，我们仅仅介绍跟PowerShell相关的部分。如果你对更深入地学习这部分技术感兴趣，我们将会提供针对后续学习的建议。

1.3 如何使用本书

本书的理念是每天完成一章的学习。我们不需要在用餐时间阅读本书，因为我们只需要接近40分钟就可以完成对一章的阅读，之后再花20分钟去享用剩余的三明治以及进行对应的练习。

主要章节

第2章至第25章为本书的主要内容，算下来差不多只要花费24顿午餐的时间来完成阅读。这也就意味着你可以在一个月内完成对本书主要章节的阅读。你需要尽可能地严格遵守制订的学习计划，不需要在既定的时间里去阅读其他章节。更为重要的是，我们需要花费一定的时间去完成每个章节之后的练习题目，用以巩固我们的学习成果。当然，并不是每个章节都需要花费完整的一小时，所以有时你在上班之前有更多的时间进行练习（或者吃午餐）。

动手实验

在主要章节的结尾都布置了需要完成的实验题目。我们会给你对应的说明，甚至可能是一两个提示，但是在本书中并不会直接给出答案。这些动手实验的答案，我们会放在MoreLunches.com上，但是建议你在查看这些答案之前尽力独立完成这部分实验。

补充资料

MoreLunches.com网站中也包含了其他一些学习资料，如额外的章节、配套视频等。实际上，每个章节至少都有一段配套视频，其中包含本章节讲解的主要内容。每段视频大概只有五分钟时间。当阅读完某章节后，你可以通过该视频回顾对应章节学习的内容。同时，你可以看到本书第一版中的视频汇总。这些视频也适用于第三版的PowerShell，并且它们都是免费的。

进一步学习

本书的某些章节仅会简单介绍一些比较酷炫的技术，在对应章节的结尾部分会给出深入学习这部分技术的建议。我们会列出额外的学习资源，包含一些免费的资料。如果你有兴趣，可以借助这些资料进行深入的学习。

补充说明

在学习PowerShell的时候，有些时候我们可能会钻入死胡同去研究为什么会这样或那样运行。如果这样学习，我们就不会学到很多实用的技能，但是我们可以对这个Shell到底是什么及其工作原理有更深入的了解。我们在“补充说明”章节中会提供这方面的信息。这些信息只需要花费几分钟就可以读完。如果你是那种喜欢钻研原理部分的人，这部分信息也可以提供一些有用的材料。如果你觉得这个小节会使得你分心而不能很好地完成实践学习，那么你可以在首次阅读时忽略这个小节。当然，如果你掌握了所有章节部分的主要内容，建议再返回阅读这部分。

1.4 搭建自己的实验环境

在本书的学习过程中，你会进行大量的PowerShell的动手实验，那么你必须构建一个属于你自己的实验环境（请记住，不要在生产环境中进行测试）。

你需要在带有PowerShell的Windows中运行本书中大部分示例以及完成每章节的动手实验。环境可以是Windows Vista，Windows 7，Windows Server 2008，Windows Server 2008 R2，Windows 8或者是

Windows Server 2012。但是需要注意的是，某些版本（如简易版）的操作系统中可能不存在PowerShell。如果你对PowerShell学习抱有很大的兴趣，那么你必须找到一个带有PowerShell的Windows系统。同时，有些动手实验是基于Windows 8 或者Windows Server 2012中PowerShell的新特性才能完成的。在每个动手实验开始时，我们都会特别说明你需要在什么操作系统中去完成这部分实践。我们建议，使用Windows 8或者Windows Server 2012去学习PowerShell，甚至你可以使用虚拟机。

在本书中，我们都是以64位（X64）操作系统为环境进行学习的。我们知道有两个版本：Windows PowerShell以及特定版本的图形化Windows PowerShell ISE。在开始菜单（Windows 8中是叫“开始”界面），这两个组件的64位版本显示为“Windows PowerShell”和“Windows PowerShell ISE”。32位版本的在快捷方式中会显示“X86”字样。在使用X86版本PowerShell时，在窗口栏中也会看到X86字样。如果操作系统本身就是32位的，那么你能只能安装32位的PowerShell，并且不会显示X86字样。

本书中的示例基于64位版本的PowerShell和对应的ISE。如果你并不是使用的64位环境，那么有些时候运行示例时可能和我们得出的结果不一致，甚至某些动手实验部分根本无法正常进行。32位版本的PowerShell主要是针对向后兼容性。例如，一些Shell扩展程序只存在于32位PowerShell中，并且也只能导入到32（或者X86）的Shell中。除非你确实需要使用这部分扩展程序，否则我们建议你在64位操作系统上使用64位的PowerShell。微软后续主要的精力会放在64位PowerShell上；如果你现在因为使用的32位操作系统而无法进行下去，那么很遗憾，以后仍然会无法继续进行。

提示： 我们完全可以在一个独立操作系统的PowerShell环境中完成本书的所有学习。但是如果使用同一个域的两台或者三台计算机的PowerShell环境联合起来进行测试，那么某些动手实验可能会变得更有趣。在本书中，我们在CloudShare.com上创建多个虚拟机来解决该问题。如果你对这种场景感兴趣，你可以了解一下这个服务或者其他类似的一些服务。但是需要注意，CloudShare.com并不是在所有国家都可以访问。

1.5 安装Windows PowerShell

从Windows Server 2008、Windows Server 2008 R2、Windows 7操作系统开始，我们已经可以使用第三版的Windows PowerShell。

Windows Vista操作系统无法支持第三版，但是可以使用第二版PowerShell。最近发布的几个操作系统中已经预装了Windows PowerShell。如果采用老版本的操作系统，那么必须手动去安装PowerShell。当然，新版本的操作系统可能会采用更新版本的PowerShell，当然这没什么坏处。

提示： 你可以采用如下方法来检查安装的PowerShell版本：进入PowerShell控制台，输入`$PSVersionTable`，然后按回车键。如果返回错误或者输出结果并未显示为“PSVersion 3.0”，那么你安装的版本就不是第三版PowerShell。

第三版PowerShell可以与第二版PowerShell安装于一台机器上，也就意味着不会损坏那些依赖于第二版PowerShell的程序。另外，我们没有必要安装第一版PowerShell，安装第三版后会自动覆盖它。最近发布的微软软件都不会依赖于第一版PowerShell。

如果你使用的是老版本的PowerShell，则需访问<http://download.microsoft.com>，然后在搜索框中键入PowerShell 3，之后根据你的操作系统选择到对应版本的PowerShell，然后进行安装。你需要找的是Windows Management Framework程序包，PowerShell是集成在这个包中进行发布的。再次申明，你需要选择到正确的版本，X86代表32位的安装包，X64代表64位的安装包。在网站上无法找到最近发布的Windows操作系统，那是因为PowerShell已经被预装到这些系统中了。

提示： PowerShell最低要求.Net Framework V4,当然如果能使用更新版本的Framework就更好了。我们建议同时最少也要安装.Net Framework 3.5 SP1以及.Net Framework 4.5版本，这样可以使PowerShell更多的功能。

安装PowerShell的同时也会安装一些配套程序，其中包含Windows 远程管理服务（WinRM），在本书后续章节中会讲到这部分。PowerShell采用类似Hotfix的方式进行安装，也就意味着安装后，也可以单独卸载。当然，一般来讲，你肯定不会希望去卸载它。PowerShell现在已经正式成为Windows 操作系统核心组件的一部分，因此对PowerShell的更新和其他Windows组件一样，以Windows的hotfix或者SP形式进行发布。

PowerShell包含两部分：基于文本的标准控制台（PowerShell.exe）和集成了命令行环境的图形化界面（ISE；

PowerShell_ISE.exe)。我们大部分时间都会使用基于文本的控制台。当然，如果你更喜欢ISE，你也可以使用。

注意： PowerShell ISE组件并没有预装到Server版操作系统中。如果你需要使用，那么你需要进入Windows的功能（使用“服务器管理器”），然后手动添加ISE功能（你也可以打开PowerShell的控制台，再执行Add-WindowsFeaturePowerShell -ise）。在未包含完整GUI模式的操作系统（如Server Core版本的系统）对应的安装程序中并没有包含ISE的安装程序。

在你继续学习PowerShell之前，建议花几分钟去设置Shell的显示界面。如果你使用基于文本的控制台，那么强烈建议你修改显示的字体为Lucida（固定宽度），不要使用默认的字体。假如使用默认字体，我们会很难去区分PowerShell使用的一些特殊字符。可以参照下面的步骤来修改显示字体。

（1）右键单击控制台界面上侧边框（PowerShell字符位于控制台界面的左上方），选择目录中的属性。

（2）在弹出的会话框中，可以在几个标签页中修改字体、窗口颜色、窗口大小和位置等。

提示： 强烈建议窗口大小和屏幕缓冲器使用相同的宽度。

另外，需要注意的是，当应用对默认控制台的修改之后，后续所有新开的窗口都会使用变更之后的设置。

1.6 在线资源

在前文中，我们已经多次提及MoreLunches.com网站，希望你有时就去访问一下。这个网站上包含如下补充资料：

- 每个章节的配套视频；
- 每章结尾动手实验的答案；
- 可供下载的代码清单（所以没有必要自己输入）；
- 额外的章节以及文章；
- 我们Windows PowerShell博客的链接地址，其中包含更多的示例以及文章；
- Don的Windows PowerShell FAQ链接地址；

- 链接到其他论坛板块的地址。在这些板块中，大家可以提问或者提交关于本书的一些反馈。

我们热衷于帮助像你这样的人来学习PowerShell，我们也在尽力提供各种不同的学习资源。同时，我们也很欢迎你给我们进行反馈，这样会促使我们找寻可以加入到我们网站的新资源，以及提升本书后续版本的质量。你可以通过MoreLunches.com网站联系到我们。

另外，也可以通过<http://jdhitsolutions.com/blog> 博客或者在Twitter上@jeffhicks找到Jeffery。

1.7 赶紧使用PowerShell吧

“可以立即使用”是我们编写本书的一个主要目标。我们在每一章中尽可能仅关注某一部分的知识，并且你在学习之后，可以立即在生产环境中使用。这就意味着，在开始的时候，我们可能会避开一些细节的讨论，但是在必要时，我们承诺后续会回到这些问题并给出详细说明。在很多情形下，我们必须在首先给出20页的理论或者直接讲解并完成某些部分的学习（暂不解释分析其中的细微差别或者详细情况）中做出选择。当需要做出这类选择时，我们总是选择第二个，以便使得你可以立即使用起来。但是之前的那些细节，我们会在另外一个时间去进行分析讲解（针对那些不会影响本书内容的小细节，我们可能会在MoreLunches.com的在线文章中去进行讲解）。

好了，背景知识大概就介绍到这里。下面就开始第2章课程的学习。

第2章 初识PowerShell

本章将协助读者选择一种最适合的PowerShell界面（不错，你可以做出选择）。如果你曾经使用过PowerShell，可以直接跳过本章，但是你阅读依旧可以从本章中找到一些对你有帮助的信息。

2.1 选择你的“武器”

微软提供了两种（如果你是很严谨的人，可以认为是四种）使用PowerShell的方式。图2.1显示了【开始】菜单中的【所有程序】界面，其中包含四种PowerShell图标。可以通过图中划线部分快速找到这些图标。

提示： 在旧版本的Windows中（本书环境基于Windows Server 2012），这些图标位于【开始】菜单中，可以通过依次选择【所有程序】>【附件】>【Windows PowerShell】来找到它们。除此之外，还可以在【开始】菜单中运行“PowerShell.exe”，然后单击【确认】，打开PowerShell的控制台应用程序。在Windows 8和Windows Server 2012中，使用Windows键（通常位于Ctrl键和Alt键之间的Windows图标）加R打开运行对话框，或者单击Windows键，然后在输入框中输入PowerShell，即可快速打开PowerShell图标。

在32位操作系统中，最多只有两个PowerShell图标。在64位系统中，最多有4个。它们分别是：

- Windows PowerShell——64位系统上的64位控制台和32位系统上的32位控制台。
- Windows PowerShell(x86)——64位系统上的32位控制台。
- Windows PowerShell ISE——64位系统上的64位图形化控制台和32位系统上的32位图形化控制台。
- Windows PowerShell(x86)——64位系统上的32位图形化控制台。



图2.1 你可以选择四种PowerShell启动方式的其中一种

换句话说，32位操作系统仅有32位的PowerShell应用程序，而64位操作系统可以有32位和64位两个版本的PowerShell应用程序，其中32位应用程序在图标名中会包含“x86”字样。需要注意的是，有些扩展程序只支持32位环境，不支持64位。微软现在已经把全部精力放到64位系统中，而32位仅用于向后兼容。

提示： 在64位系统中，人们经常会错误地打开32位应用程序，此时应该注意窗体的标题，如果显示“x86”，证明你在运行32位程序。另外，64位扩展程序不能运行在32位应用程序中，所以建议用户把64位应用程序以快捷方式的形式固定在开始菜单中。

2.1.1 控制台窗口

图2.2展示了PowerShell控制台窗口界面，这是大多数人第一次见到的PowerShell界面。

接下来从使用简单的PowerShell控制台命令和参数开始本小节。

- PowerShell不支持双字节字符集，也就是说，大部分非英语语言不能很好地展示出来。
- 剪切板操作（复制和粘贴）使用的是非标准键，意味着使用起来较为不便。
- PowerShell在输入时会提供少量帮助信息（这个相对于ISE而言，在下面即将介绍）。

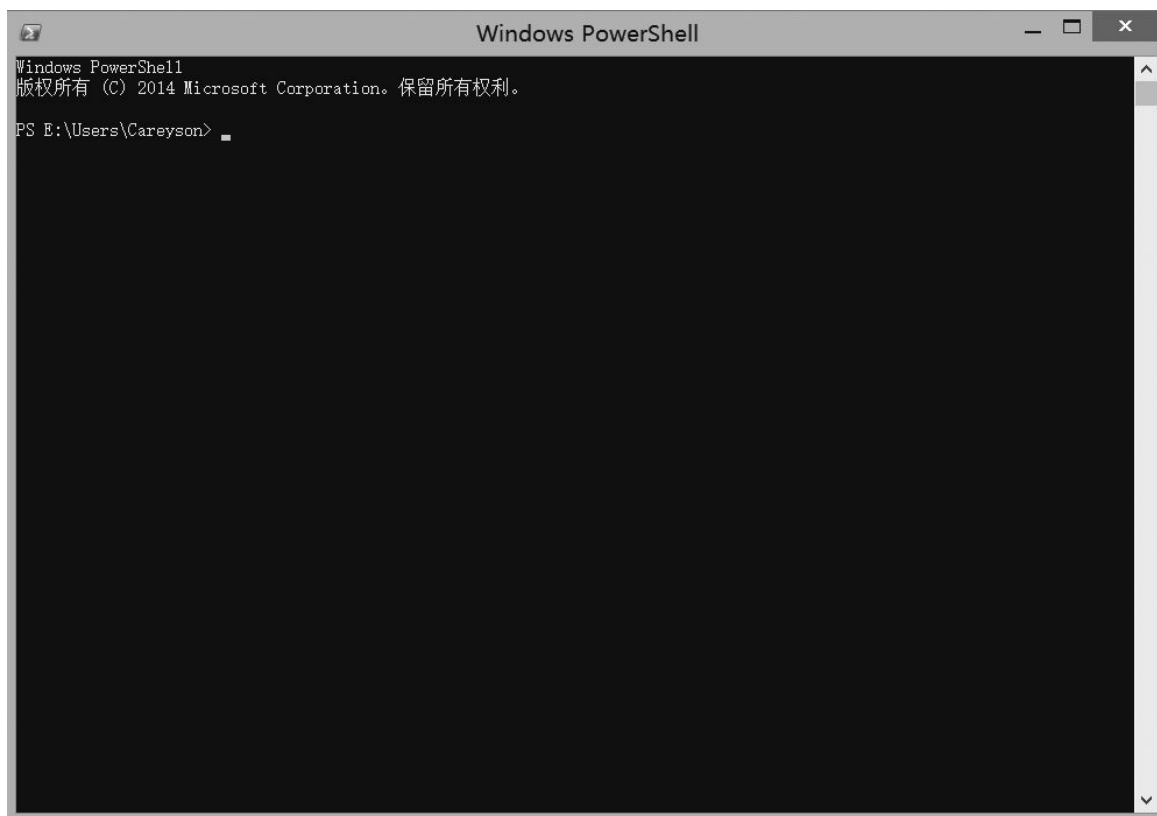


图2.2 标准的PowerShell控制台窗口：PowerShell.exe

综上所述，PowerShell控制台应用程序将是你没有安装GUI Shell的服务器上运行PowerShell的唯一选择（如一些“服务器核心功能”安装或者Windows Server中服务器GUI Shell功能被移除或没有安装的情景）。其优点是：

- 控制台程序非常轻量，可以快速加载且不需要太多内存。
- 不需要任何非PowerShell自身必需的.NET Framework之外的资源。
- 可以在黑色背景中设置绿色字体，正如在20世纪70年代的机器上工作一样。

如果你打算使用控制台应用程序，在你配置时会有些建议可供参考。可以通过单击窗体左上角的图标，并选择【属性】来实现，如图2.3所示。

在【选项】标签页，可以调大“命令记录”的缓冲区大小。这个缓冲区可以记住你在控制台输入的命令，并且通过键盘的上、下键重新调用它们。你也可以通过按F7键来弹出命令列表。

在【字体】标签页，选择稍微大于默认12像素的字体。不管你是否拥有1.5的视力，稍微提高一下字体大小也没什么坏处。PowerShell希望你能在

大量类似的字符中快速区分它们，比如’（撇号或单引号）和`（重音符）。如果使用小像素字体，识别这类字符将比较困难。



图2.3 配置控制台应用程序的属性

在【布局】标签页，把所有“宽度”设为相同的数值，并且确保结果窗体能适合你的显示屏。如果设置不合理，会导致PowerShell窗体下方出现水平滚动条。这可能导致一部分输出结果被挡住，从而忽略了它们的存在。作者的学生就曾经花了半小时来运行命令，但是却完全没有输出，实际上输出被隐藏在右边。

最后，在【颜色】标签页，强烈建议不要修改，保持高度反差将有助于阅读。如果你不喜欢默认的蓝底白字，可以考虑中灰底黑字的形式。

需要记住一件事：这个控制台应用程序并不是真正的PowerShell，仅仅是你和PowerShell交互的界面。控制台应用程序本身可以追溯到大约1985年，所以你不要指望能从中得到流畅的体验。

2.1.2 集成脚本环境（ISE）

图2.4展示了PowerShell 集成脚本环境，也称ISE。

提示：如果你不经意打开了普通的控制台应用程序，可以输入“ise”并按回车键，从而打开ISE。

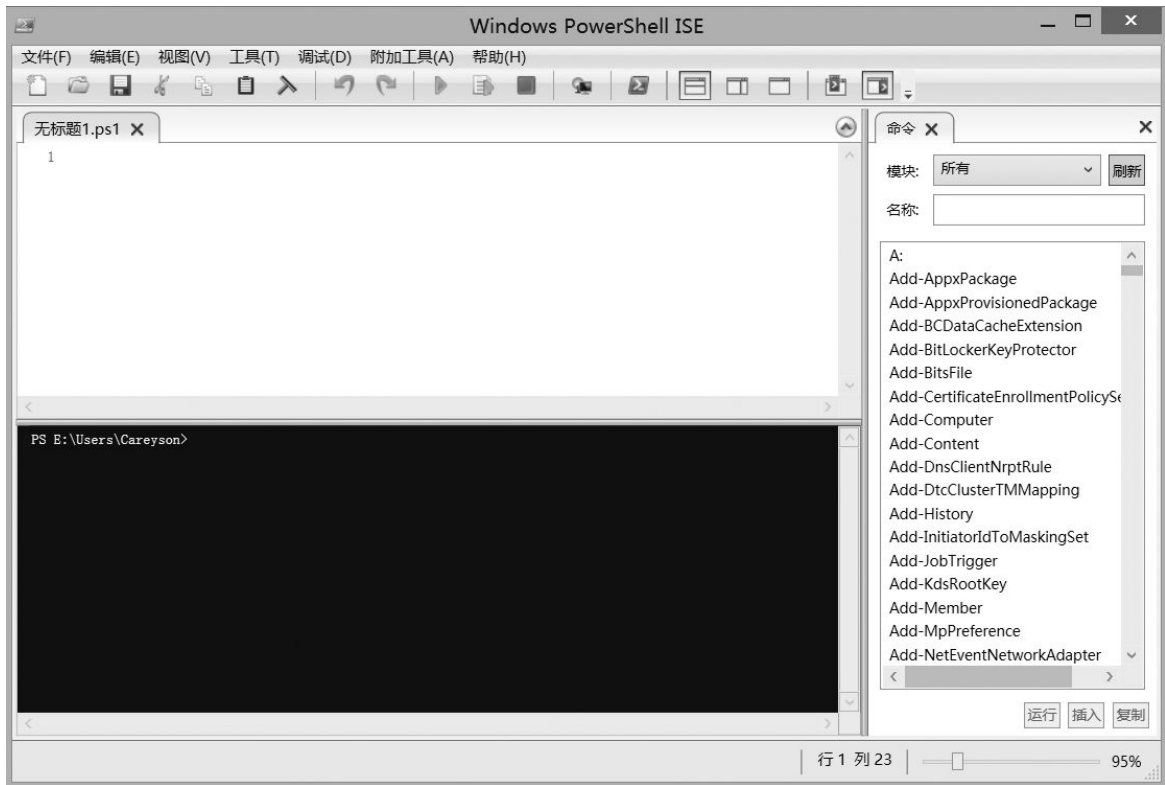


图2.4 PowerShell ISE（PowerShell_ISE.exe）

表2.1 ISE的优缺点

优点	缺点
ISE界面友好且支持双字节字符集	ISE要求Windows Presentation Foundation（WPF），意味着不能在没有安装GUI的服务器上运行ISE
在后续章节可以看到ISE能在你创建PowerShell命令和脚本时提供更多的帮助	启动和运行需要较长时间，但是这通常只是几秒的差异
ISE使用常规的复制、粘贴按键	ISE不支持转录

表2.1列出了ISE的优缺点，从中可以得到大量背景信息。

下面从一些基本定位开始。图2.5展示了ISE的三个主要区域，图中划线部分即为ISE的工具栏。

在图2.5中，最上方的区域是【脚本编辑窗格】，直到本书最后才会用到。在它的右上角，可以看到一个蓝色的小箭头，单击它可以最小化【脚本编辑窗格】并最大化【控制台窗格】。控制台窗口是我们将要使用的地方。右边是【命令管理器】，可以通过它最右上方的“X”打开或者关闭这个窗口。除此之外，可以通过工具栏倒数第二个按钮来浮动【命令管理器】。如果你已经关闭【命令管理器】又想要它重新出现，可以单击工具栏的最后一个按钮。工具栏中的前三个按钮用于控制【脚本编辑器】和【控制台窗格】的布局。可以通过这些按钮把窗体设置为【在顶部显示脚本窗格】、【在右侧显示脚本窗格】和【最大化显示脚本窗格】。

在ISE窗口的右下角，可以发现用于改变字体大小的滚动条。在【工具】菜单中，可以找到一个【选项】项用于配置定制化的颜色方案和其他显示配置——这完全根据你的喜好而定。

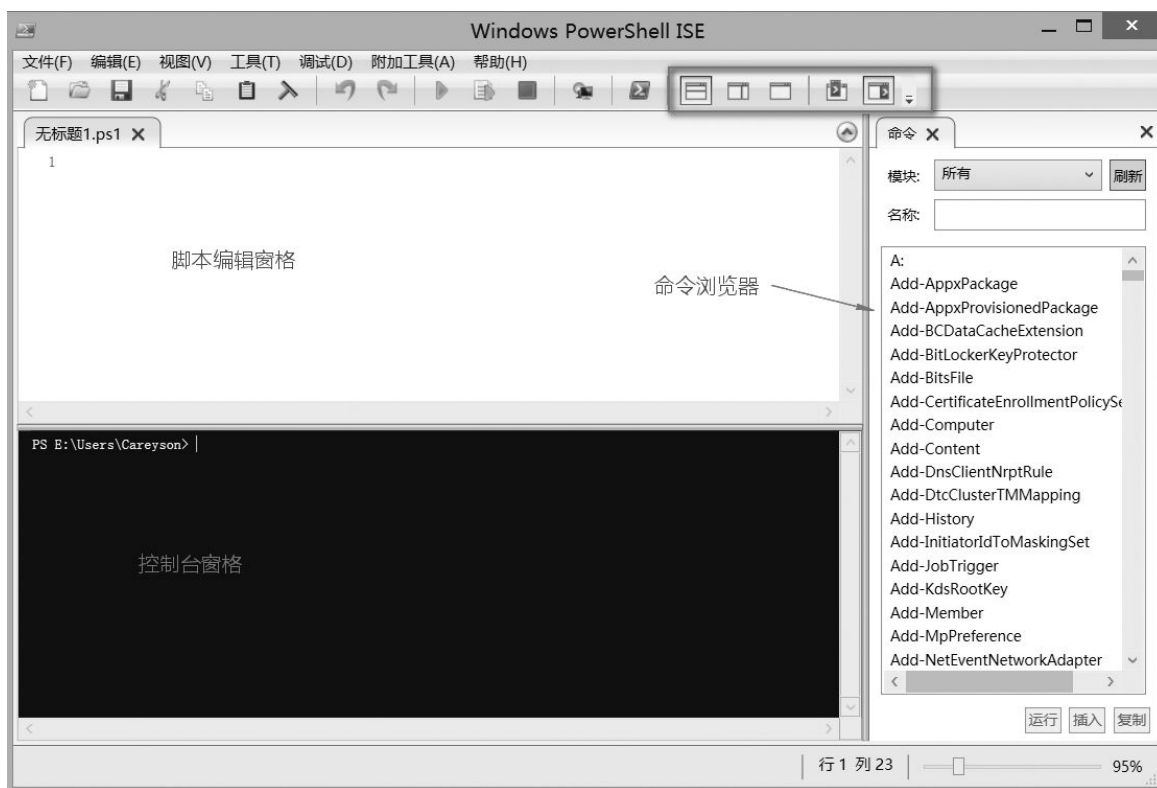


图2.5 ISE的三个主要区域及控制它们的工具栏

动手实验：首先我们假设读者将会在余下章节中只使用ISE，然后隐藏【脚本编辑窗格】。如果你愿意，也可以把【命令管理器】隐

藏。把字体大小设置到你喜欢的样子。如果你不能接受默认的颜色方案，请自行选择。如果你更喜欢控制台应用程序，请放心使用，本书的绝大部分内容同样能在控制台中运作。一些仅在ISE中才能使用的功能将会额外标注。

2.2 重新认识代码输入

PowerShell是一个命令行接口，意味着你需要大量输入代码。然而输入命令就意味着可能出现错误，例如拼写错误。幸运的是，所有PowerShell应用程序都提供了最小化错别字的方式。

动手实验： 接下来的例子在本书中可能显得不太实际，但是在本节看来却很炫。读者可以在自己的环境中尝试一下。

控制台应用程序支持四种“Tab键补全”。

- 输入“Get-S”，然后按几下“Tab”键，再按Shift+Tab组合键。PowerShell会循环地显示以Get-S开头的所有命令。然后不停按Shift+Tab组合键，直到出现你期望的命令为止。
- 输入“Dir”，按空格键，然后输入C:\，再按“Tab”键，PowerShell会从当前文件夹开始循环遍历所有可用的文件和文件夹名。
- 输入“Set-Execu”，按“Tab”键，然后输入一个空格和横杠（-），再开始按“Tab”键，可以看到PowerShell循环显示当前命令的所有可用参数。另外，也可以输入参数名的一部分，如-E，然后按“Tab”键，开始循环匹配的参数名。按“Esc”键可以清空命令行。
- 再次输入“Set-Execu”，按“Tab”，再按空格键，然后输入-E，再次按“Tab”键，然后按一次空格键，再按“Tab”键。PowerShell会循环显示关于这些参数的合法值。这个功能仅对那些已经预设了可用值（称为枚举）的参数有效。按“Esc”键同样可以清空命令行。

PowerShell ISE提供了类似功能，甚至可以说比“Tab补全”功能更好的功能——智能提示。这个功能在前面四个情况下都能运作。图2.6演示了如何通过弹出菜单来实现你在使用“Tab”键时完成的功能。可以使用上下箭头按钮来滚动菜单，找到你想要的选项，然后按“Tab”或者按“Enter”键来选择，再继续输入剩余代码。

智能提示可以工作在ISE的控制台窗格和脚本编辑窗格中。

警告： 当你在PowerShell中输入时，请极其小心。在某些情况下，一个错位的空格、引号或者单引号都会带来错误或者失败。如果出现了错误，请再三检查你的输入内容。

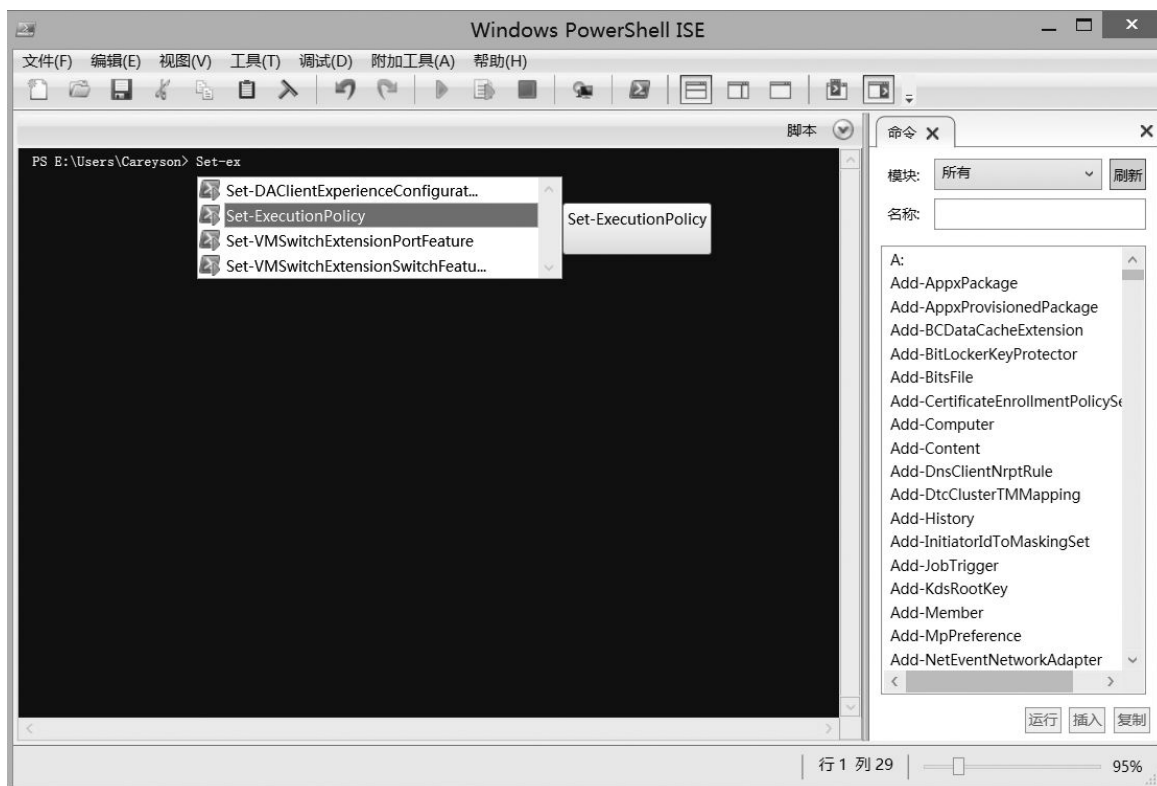


图2.6 在ISE中类似Tab自动补全功能的智能提示功能

2.3 常见误区

接下来让我们快速回顾一些会影响你享受PowerShell旅途的绊脚石。

- 在控制台应用程序中的水平滚动条——从多年的教学经验中我们得知，正如前面提到过，配置控制台的窗口，使其不出现水平滚动条是非常重要的。
- 32位 vs 64位——建议你使用64位的Windows并使用64位的PowerShell应用程序（没有出现“x86”字样的应用程序）。虽然对于某些人来说，购买64位的计算机和64位的Windows可能是件大事，但是如果你希望PowerShell高效运行，那么这些投资还是必须的。虽然在本书中我们尽可能覆盖32位环境，但是这些内容在64位的生产环境上将带来很大的差异。
- 确保PowerShell应用程序的窗体标题显示“管理员”——如果你发现打开的窗体上没有“管理员”字样，关闭窗体并右键单击PowerShell图标，选择“以管理员身份运行”。在生产环境中，不一定总是要这样。本书后面将演示如何使用特定的凭据运行命令。但是通常情况下，为了避免运行时出现一些问题，最好确保以管理员身份运行PowerShell。

2.4 如何查看当前版本

在很大程度上，找出当前使用的PowerShell版本不是件容易的事，因为每个发布版本都安装在“1.0”的目录下面（1.0是引用的Shell引擎语言版本，即所有版本都向后兼容到v1）。针对PowerShell，有一种简单的方式检查：

```
PS C:\> $PSVersionTable
Name                           Value
----                           -
PSVersion                     3.0
WSManStackVersion             3.0
SerializationVersion          1.1.0.1
CLRVersion                    4.0.30319.17379
BuildVersion                   6.2.8250.0
PSCompatibleVersions           {1.0, 2.0, 3.0}
PSRemotingProtocolVersion      2.2
```

输入\$PSVersionTable，然后按“Enter”键。可以看到每个PowerShell相关技术的版本号，包括PowerShell自身的版本号。如果命令不能运行，或者显示最少需要PSVersion 3.0等字样，则需要使用第1章中展示的方式安装最少3.0的版本。

动手实验： 现在就开始使用PowerShell，首先检查你的PowerShell版本是否达到最少3.0版本，如果不是，请先升级到最少3.0版本。

2.5 动手实验

因为这是本书第一个实验，所以我们会花一些时间去描述它们是如何运作的。对于后续的实验，我们会给出一些任务，以便你可以自己动手尝试和完成。一般我们只给出一些提示或者方向指引。所以从现在开始，你只能靠自己了。

我们保证所有需要用于完成实验的知识仅限于当前章节或之前的章节（对于之前章节的知识，我们一般采用提示的方式给出）。我们不会把答案说得太明显，更多地，当前章节会告诉你如何发现你所需要的信息，你需要自己去发现这些问题的答案。虽然看起来有点让人沮丧，但强迫自己去完成，从长远来说绝对可以让你在PowerShell的世界里面走得更好。

顺带提醒，你可以在MoreLunches.com中找到一些示例答案。这些答案不一定完全匹配你的问题，但是当我们一步一步地深入之后，答案将变得越来越准确。实际上，我们会发现PowerShell针对几乎所有的问题都能提供几种甚至更多的方式实现。我们会尽可能地使用最常用的方式，但是如果你尝试使用另外一些不同的方式，并不代表你是错误的。任何能实现结果的方式都是正确的。

注意： 本实验需要PowerShell v3或以上版本。

我们从简单的例子开始：希望你能从控制台和ISE的配置中实现相同的结果。然后按照下面五步进行。

1. 选择适合你自己的字体和颜色。
2. 确保控制台应用程序下方没有水平滚动条。（本章中已经第三次提到，可见其重要性。）
3. 在ISE中，最大化控制台窗格，移除或最小化命令管理器。
4. 在所有应用程序中，输入一个单引号“'”和一个重音符“`”，确保你可以轻易识别出来。在美式键盘中，重音符位于左上角，在“Esc”键下面，和波浪号“~”位于同一个键中。
5. 同样输入括号（），中括号[]，尖括号<>和花括号{}，确保你所选择的字体和大小能很好地展示这些符号，足以让你马上识别出来。否则，请选择其他字体或者加大字体大小。

前面已经提到过这些步骤，所以你对此应该没有任何疑问，你要做的只是一步一步地完成。

2.6 进一步学习

除了微软提供PowerShell之外，你会发现其他针对PowerShell定制的自由或商用的编辑工具。下面提供常见的几种。你可以去试用。就算商业版也会有试用期，所以不妨去尝试一下。

- *SAPIEN PrimalScript* 和 *PrimalForms* ——来自<http://primaltools.com> 的两个商业版工具。
- *Idera PowerShell Plus* ——来自<http://idera.com> 的编辑器与控制台环境。

你可能找到其他工具，但是这四种工具的用户群是最多的，而且是我们最常用的工具。我们和这些公司没有任何关系（仅仅是欣赏他们的成果）。经常有人问，这些软件里面哪个用得最多。此时我们只能说使用ISE最多，因为我们经常重建虚拟机，并且懒得重新安装这些编辑器（而且我们也没有时间去为此专门写一个PowerShell脚本）。即便如此，当我们的确要确定使用一个第三方工具时，通常是PowerShell Plus，因为我们喜欢它提供的增强版控制台而不仅仅是脚本编辑器。对于这种集成，我们深表欢迎。但是还是建议读者根据需求和预算选择这些工具。

2012年1月，Don评论了各种各样的PowerShell环境。假如你在2014年1月开始阅读本书（此时文章可能过时，并且根据当前可用产品而更新），可以从下面链接中查看：<http://ConcentratedTech.com>。它成文于PowerShell v3时代，并且适用于文章中提到的产品。不管怎样，它还是可以作为学习Shell脚本的不错的起点。

第3章 使用帮助系统

在这本书的第1章，我们提到由于图形用户界面具有更强的可发现性，所以更容易学习和使用。但对于像PowerShell这样的命令行接口-CLIs（command-line interfaces）的学习却往往要困难一些，因为它们缺乏可发现性这个特性。事实上，PowerShell拥有出色的可发现性，但是它们并不是那么明显。其中一个主要的可发现性的功能是它的帮助系统。

3.1 帮助系统：发现命令的方法

请忍受1分钟的时间让我们走上讲台给你讲述下面的内容。

我们工作在一个不是特别重视阅读的行业，但是我们有一个缩写RTFM（Read The Friendly Manual）。当我们希望他们可以“阅读易于使用的手册”时，就能巧妙地把命令传递给用户。大多数管理员更加倾向于直接上手、依赖于GUI工具的提示、上下文菜单等这些GUI的可发现性工具来领会如何操作。这也是我们工作的方式。我们假设你也是以同样的方式进行工作的。但是我们来认清一件事情：

如果你不愿意花时间去阅读PowerShell的帮助文档，那么你就无法高效使用PowerShell，也很难进一步学习如何使用它，更不用说使用它管理类似Windows或Exchange等产品，最终你无法摆脱使用GUI的方式。

让我们澄清一下，虽然上面一段看上去很蠢，但绝对是真理。想象一下，当你使用活动目录和计算机或是其他管理控制台时没有帮助提示、菜单、上下文菜单会怎么样。好比学习PowerShell而不去花时间去学习帮助文件也是如此。这就好像你去宜家不阅读手册就去组装家具，那么你必然会经历挫折、困惑以及感到无能为力。为什么呢？

- 如果你需要执行一项任务，但是却不知道应该使用什么命令，帮助系统可以帮助你找到这个命令，而不是使用Google或者Bing。
- 如果你在运行一个命令的时候返回错误信息，帮助系统可以告诉你如何正确运行命令而不出现错误。

- 如果你想将多个命令组合在一起执行一项复杂的任务，帮助系统可以帮你找到哪些命令是可以和其他命令结合使用。你不需要在Google或者Bing搜索示例，只需要学习它们是怎么使用的，以便你可以创建出自己的示例和解决方案。

我们意识到我们的讲述过于强调帮助的重要性，但我们看到学生在课堂上或者在工作中面临的问题：如果他们能腾出几分钟坐下来、深呼吸和阅读帮助，90%的问题都能得到解决。阅读这一章，将帮助大家理解正在阅读的帮助文档。

从现在开始，我们来介绍几个鼓励你阅读帮助文档的原因。

- 虽然我们将在我们的示例中向你展示许多命令（我们几乎从未展示一个命令的完整功能和选项），但是你也应该阅读我们展示每个命令的帮助，这样你才会熟悉每个命令所能够完成的额外工作所能够完成的。
- 在本书的实验里，我们将提示你使用什么命令来完成任务，但是我们不会提示语法细节。为了完成这些实验，你必须自己使用帮助系统来找到相应命令的语法。

我们向你保证，掌握帮助系统是成为PowerShell专家的一个关键。但你不会在帮助文档中找到每一个细节。很多高级资料并没有记录在帮助系统，但为了有效的日常管理，你需要熟练运用帮助系统。本书会帮助你深入理解该系统，并和内置帮助结合使用，可以教会你在帮助文档中没有具体解释的部分。

是时候走下讲台了。

Command对比Cmdlet

PowerShell 包含了很多不同类型的可执行命令，有些叫作Cmdlet，有些叫作函数，还有一些被称为工作流，等等。它们的共同点都是命令，帮助系统中都对它们进行了展示。每个Cmdlet在PowerShell中都是唯一的，你运行的大多数命令都属于Cmdlet。但在谈论一般类的可执行程序的时候，我们会使用“命令”来表示，从而保证一致性。

3.2 可更新的帮助

当你第一次使用帮助时，你也许会很惊讶，因为里面什么都没有。不要着急，我们会为你讲解。

微软在PowerShell v3中加入了一个新的特性，叫作“可更新的帮助”。PowerShell可以通过互联网进行下载更新、修正和扩展。

不过，为了做到可更新，微软不能把任何帮助放到安装包中。当你需要查看一个命令的帮助时，你可以得到一个自动生成的简易版的帮助，还可以通过这些信息来提示你怎么更新帮助文档，类似下面的信息。

```
PS C:\> help Get-Service
```

NAME

Get-Service

SYNTAX

```
Get-Service [[-Name] <string[]>] [-ComputerName <string[]>]
[-DependentServices] [-RequiredServices] [-Include <string[]>]
[-Exclude <string[]>] [<CommonParameters>]
```

```
Get-Service -DisplayName <string[]> [-ComputerName <string[]>]
[-DependentServices] [-RequiredServices] [-Include <string[]>]
[-Exclude <string[]>] [<CommonParameters>]
```

```
Get-Service [-ComputerName <string[]>] [-DependentServices]
[-RequiredServices] [-Include <string[]>] [-Exclude <string[]>]
[-InputObject <ServiceController[]>] [<CommonParameters>]
```

别名

gsv

备注

Get-Help 在本机无法找到关于这个Cmdlet命令对应的帮助文档。
这只显示了部分帮助信息。

-- 可使用**Update-Help**下载和安装包含这个Cmdlet模板的帮助文档。

-- 可输入"**Get-Help Get-Service -Online**"命令或者

输入网址<http://go.microsoft.com/fwlink/?LinkID=113332>

查看关于帮助主题的在线文档。

提示： 如果你的本机没有安装帮助，在你第一次使用帮助的时候，PowerShell 会提示你更新帮助系统。

更新PowerShell的帮助文档应该是你的首要任务。这些文件存储在System32这个目录下，这意味着你的Shell必须在更高特权下运行。如果在PowerShell 的标题中没有出现“管理员”的字眼，你将会得到一个错误信息：

```
PS C:\> update-help
Update-Help : 无法更新以下模块的帮助：
'Microsoft.PowerShell.Management, Microsoft.PowerShell.Utility,
Microsoft.PowerShell.Diagnostics, Microsoft.PowerShell.Core,
Microsoft.PowerShell.Host, Microsoft.PowerShell.Security,
Microsoft.WSMan.Management' :
命令无法更新 Windows PowerShell 核心模块或 $psHOME\Modules 目录中任意模
块的帮助主题。若要更新这些帮助主题，请使用“以管理员身份运行”命令启动Windows
PowerShell，然后重试运行 Update-Help

。
所在位置 行:1 字符: 1
+ update-help
+ ~~~~~
+ CategoryInfo          : InvalidOperation: (:) [Update-
Help], Except
ion
+ FullyQualifiedErrorId :
UpdatableHelpSystemRequiresElevation, Micros
oft.PowerShell.Commands.UpdateHelpCommand
```

前面错误信息中以粗体进行标识的部分，告诉你问题的所在并告诉你如何解决它。以管理员身份来运行Shell，再次运行Update-Help命令，这样它就可以运行了。运行需要花费几分钟的时间。

每隔一个月左右的时间重新获取帮助是一个很重要的习惯。PowerShell 甚至可以下载微软之外的命令帮助文档，只要这些命令模块在合适的位置进行本地化之后加入到在线上以供下载。

假如你的计算机不能连上互联网，那该怎么办呢？不要担心，首先找到一台可以上网的机器，并使用**Save-Help**命令把帮助文档下载一份到本地。然后把它放到一个文件服务器或者其他你可以访问的网络中。接着通过在**Update-Help**加上**-SourcePath**参数指向刚刚下载的那份帮助文档的地址。这可以让局域网内任何计算机从中心服务器获取更新后的帮助，代替从互联网获取。

3.3 查看帮助

Windows的PowerShell 提供了**Get-Help**这个Cmdlet命令来访问帮助系统。你可能看到很多示例（特别是在互联网）都是使用“**Help**”或“**Man**”这个关键字来代替**Get-Help**。**Man**和**Help**根本都不是原始的Cmdlet命令，而是对核心Cmdlet命令进行包装的函数。

Help的工作原理跟**Get-Help**是一样的，但它可以把输出的信息通过管道传送给**More**命令。这样你就可以以分屏这样友好的方式来查看帮助的内容，而不是一次性打印出所有的帮助信息。运行**Help Get-Content**和 **Get-Help Get-Content**，将会返回相同的结果。前者是一次一页显示，你也可以使用**Get-Help Get-Content | More**分页显示，但这需要输入更多的字符，我们使用**Help**就能完成输入了。我们讲了这么多的目的是想让你知道隐藏在下面真实的东西。

注意：从技术上来说，**Help**属于一个函数，而**Man**属于**Help**的一个别名，或者叫昵称。但是它们返回的结果都是一样的。我们将会在下一章讨论别名。

顺便提醒一下，有些时候你可能会讨厌分页显示，因为你想一次性获取所有的信息，但是它却一次次让你输入空格键来显示剩下的信息。如果你遇到这样的情况，在Shell控制台窗口按**Ctrl+C**组合键来取消命令并立即返回到Shell。**Ctrl+C**组合键总是表示“返回”的意思，而不是“拷贝到剪切板”的意思。而在图形化Windows PowerShell ISE中，**Ctrl+C**表示拷贝到剪切板。工具栏中有一个红色按钮“停止”，它可以用于停止正在运行的命令。

注意：很多命令在图形化的ISE中不起作用，甚至当你使用**Help**或**Man**时，它会一次性返回所有的帮助信息，而不是一次返回一页。

帮助系统有两个主要的目标：一个是帮助你找到特定任务的命令，另一个就是帮助你学会使用找到的这些命令。

3.4 使用帮助找命令

从技术上来说，帮助系统不知道Shell中存在哪些命令。它只知道有哪些可用的帮助主题。某些命令可能并没有帮助文档，这会导致帮助系统不能确认这个命令是否存在。幸好微软几乎每个发布的Cmdlet都包含一个帮助主题，这意味着你通常不会发现不同。另外，帮助系统也包含了除特定Cmdlet之外的其他信息，包括背景概念和其他基础信息。

跟大多数命令一样，Get-Help（等同于Help）有几个参数。其中一个最为重要的参数是-Name。这个参数指定你想要访问帮助的主题名称，并且它是一个定位参数，所以你不用必须输入-Name，可以只提供需要查看命令的名称。它也支持通配符，这让帮助系统更加容易找到命令。

例如，你想操作系统事件日志，但是你却不知道使用什么命令，你决定搜索帮助主题包含的事件日志，可以运行下面两个命令中的任何一个。

```
Help *log*
Help *event*
```

第一个命令在你的计算机返回如下列表：

Name	Category	Module
----	-----	-----
Clear-EventLog	Cmdlet	
Microsoft.PowerShell.M...		
Get-EventLog	Cmdlet	
Microsoft.PowerShell.M...		
Limit-EventLog	Cmdlet	
Microsoft.PowerShell.M...		
New-EventLog	Cmdlet	
Microsoft.PowerShell.M...		
Remove-EventLog	Cmdlet	
Microsoft.PowerShell.M...		
Show-EventLog	Cmdlet	

Microsoft.PowerShell.M...		
Write-EventLog	Cmdlet	
Microsoft.PowerShell.M...		
Get-AppxLog	Function	Appx
Get-DtcLog	Function	MsDtc
Reset-DtcLog	Function	MsDtc
Set-DtcLog	Function	MsDtc
Get-LogProperties	Function	PSDiagnostics
Set-LogProperties	Function	PSDiagnostics
about_Eventlogs	HelpFile	
about_Logical_Operators	HelpFile	

注意： 你可以注意到，前面的这个列表的Appx、MsDtc模块包含命令（和函数）等。即使你还没有加载这些模块扩展到内存，帮助系统也一样会显示所有模块。这可以帮助你发现电脑上被遗漏的命令。它可以发现那些安装在适当位置所有扩展中的命令。对此，我们会在第7章进行讨论。

前面的列表中有许多关于事件日志的函数，它们都基于“动词-名词”这个命名格式，但是最后出现了两个关于帮助主题的特殊Cmdlets命令。这两个“about”主题提供了关于某个命令的背景信息。最后一个看起来跟事件日志没有什么关系，但是它被搜索到是因为其中有一个单词“logical”的其中一部分包含了“log”。只要有可能，我们尽量使用“*event*”或者“*log*”来搜索，而不是使用“*eventlog*”，因为我们可以返回尽可能多的结果。

一旦发现从名称看起来可能是你所需的Cmdlet时（比如说，后面示例中Get-EventLog看起来就是做这件事的），你就可以查看该Cmdlet的帮助文档进行确认。

```
Help Get-EventLog
```

不要忘记使用Tab键来补全命令！它可以让你只输入部分命令名，按下Tab键，接着Shell会完成与你刚刚输入最为接近匹配的命令。你可以连续按Tab键来选择其他匹配的命令。

动手实验： 输入Help Get-Ev，接着按下Tab键。第一次匹配到的是Get-Event，这并不是你想要的；再次按下Tab键就匹配到

Get-EventLog，这就是你想要的命令。你可以敲回车键来确认这个命令并显示这个命令对应的帮助信息。如果你使用ISE，你不需要敲Tab键。所有匹配的命令都会以列表的形式呈现，你可以选择其中一个并敲回车键，这样就完成了命令的输入。

你也可以使用最为重要的“*”通配符，它可以在**Help**后面占用零个或多个字符。如果PowerShell只找到一个匹配你输入的命令，它并不是以列表的形式返回这个单一项，而是直接显示这一单项的具体帮助内容。

动手实验： 运行**Help Get-EventL***命令，你应该可以看到关于**Get-EventLog**的帮助信息，而不是返回一个匹配的帮助主题列表。

如果你一直跟随本书的示例进行实验，那么现在你就应该在看**Get-Eventlog**的帮助文档了。这个文档被称为概要帮助，这意味着它只有简单的命令描述和语法提示。这些信息是非常有用的，当你需要快速认识一个命令的使用，并且我们通过这个帮助文档来进行示例讲解。

补充说明

有些时候，我们想分享的信息虽然不错，但不是至关重要的Shell知识。我们将把这些信息放到“超越自我”部分，正如现在这个部分。如果你跳过这段，你并没有什么损失，但是如果你阅读了，你通常会学会以另外一种方式来解决这个问题，或者得到额外深入理解PowerShell的机会。

我们前面提到过**Help**命令并不是为了搜索**Cmdlet**命令，而是为了搜索帮助主题。因为每个**Cmdlet**都有一个帮助文件，我们可以说，这些搜索是检索相同的结果集。但是你也可以直接使用**Get-Command**命令来搜索**Cmdlet**命令（或者它的别名**Gcm**）。

跟**Help**这个命令一样，**Get-Command**接受通配符，意味着你可以运行**Gcm *event***来查看所有名称包含“event”的命令。不管怎么样，这个返回的列表将包含不止**Cmdlet**命令，还会包含一些不一定有用的外部命令，如**netevent.dll**。

一个比较好的方式是使用“-名词”或者“-动词”参数。因为只有 Cmdlet 名的名称有名词和动词，返回的结果将会限制为 Cmdlet 命令。Get-Command -noun *event* 将会返回一个关于事件命令的列表；Get-Command-verb Get 将会返回一个具有检索能力的列表。你也可以使用 -CommandType 参数来指定命令的类型。比如，Get-Command *log* -type Cmdlet 将会返回一个所有命令名称包含“log”的命令列表，并且这个列表不会包括任何其他扩展应用程序或者扩展命令。

3.5 详解帮助

PowerShell 的 Cmdlet 帮助文件有一些特殊的约定。从这些帮助文件中提取大量信息的关键是你需要明白自己在寻找的是什么，并学会更高效地使用这些 Cmdlet 命令。

3.5.1 参数集和通用参数

大部分命令可以有很多不同的使用方式，这依赖于你需要用它们来干什么。例如，下面是 Get-EventLog 的语法帮助部分。

```
SYNTAX
    Get-EventLog [-AsString] [-ComputerName <string[]>] [-List]
    [<CommonParameters>]

    Get-EventLog [-LogName] <string> [[-InstanceId] <Int64[]>] [-
    After <DateTime>]
    [-AsBaseObject] [-Before <DateTime>] [-
    ComputerName<string[]>] [-EntryType
    <string[]>] [-Index <Int32[]>] [-Message<string>] [-Newest
    <int>] [-Source
    <string[]>] [-UserName <string[]>] [<CommonParameters>]
```

注意，这个命令在语法部分出现了两次，这表示这个命令提供了两个不同的参数集，你可以有两种方式来使用这个命令。你可能已经注意到，有些参数是这两个参数集共享的。例如，这两个参数集都包含 -ComputerName 参数。但是这两个参数集总是会有些差异。在这个实例中，第一个参数集提供了 -AsString 和 -List，这两个参数都没有出

现在第二个参数集中；而第二个参数集包含许多第一个参数集中没有的参数。

下面来说明它们是如何工作的：如果你使用一个只包含在某个参数集中的参数，那么你就只能使用同一个参数集里的其他参数。如果你选择使用-List参数，那么你能使用的其他参数就只能是-AsString和-ComputerName，因为存在-List的参数集中只剩这两个参数可以选择了。你不能添加-LogName参数，因为它不存在于第一个参数集中。这意味着-List和-LogName是相互排斥的，即你不能同时使用它们，因为它们存在于不同的参数集中。

有些时候，可以只运行命令参数集中共同的参数部分。在这种情况下，Shell通常会选择第一个参数集。明白你运行的命令属于哪个参数集是非常重要的，因为每个参数集意味着不同的功能。

你可能已经注意到，在每个PowerShell的Cmdlet参数的结尾都有[<CommonParameters>]。不管你以何种方式使用Cmdlet，这泛指每个Cmdlet命令都是使用的一组包含8个参数的集合。现在暂时不讨论通用参数，我们会在本书后面章节真正使用它们的时候来讨论。不过，在本章后面，如果你有兴趣，我们会告诉你哪里可以学习到更多关于通用参数的知识。

谨记：如果你访问<http://MoreLunches.com>，并在首页搜索这本书（译者注：本书的英文名），你会获得各种免费配套材料。这些材料包含主要概念的示例视频，如参数和参数集，这可以帮助你更加容易理解它们。

3.5.2 可选和必选参数

运行一个Cmdlet命令，你不需要提供全部参数。PowerShell的帮助文档把可选参数放到一个方括号中。例如，[-ComputerName <string[]>]表示整个-ComputerName参数是可选的。你可以根本不使用它，因为在没有为这个参数指定一个具体值的时候，Cmdlet会默认为本地计算机。这也就是为什么[<CommonParameters>]在方括号内，你就可以在不使用任何通用参数的情况下运行这个命令。

几乎所有的Cmdlet命令都最少有一个可选参数。你可能永远不会需要使用其中的一些参数，你或许只需要使用其他日常参数。记住，

当你选择一个参数时，你只需输入足够的参数名称就可以让PowerShell明确找出参数的意思。例如，**-L**不能充分表示**-List**，因为**-L**可以表示**-LogName**。但是**-Li**会是**-List**的一个合适的缩写，因为其他参数没有以**-Li**开头的。

如果你在运行命令但忘了指定必选参数，会发生什么事情呢？来看看**Get-EventLog**的帮助。例如，你可以看到**-LogName**参数是具有强制性的，这个参数不是以方括号结束的。尝试在没有指定日志名称的情况下运行**Get-EventLog**。

动手实验：通过运行没有任何参数的**Get-EventLog**命令来查看这个例子。

PowerShell会提示你需要强制输入**LogName**参数。如果你输入类似**System**或者**Application**的参数值之后敲回车键，这个命令就能正常运行了。你可以按下**Ctrl-C**组合键来终止这个命令。

3.5.3 定位参数

PowerShell设计师知道有些参数会被频繁地使用，而你不希望不断地输入参数名。通常来说，参数是具有位置性的。这意味着只要你把参数值放在正确的位置，你就可以只提供这个参数值，而不需要输入具体的参数名。

有两种方式可以用来确定定位参数：通过语法概要或者通过详细的帮助文档。

在语法概要中找到定位参数

你可以在语法概要中找到第一种方式：只有参数名被方括号括起来的参数。比如，查看**Get-EventLog**的第二个参数集的前两个参数：

```
[ -LogName ] <string> [ [ -InstanceId ] <Int64[]> ]
```

第一个参数：**-LogName**。它是必选的。我们可以识别出它是必选参数，是因为它的参数名和参数值不在一个方括号里面。但是它的参

数名处在一个方括号内，这让它成了一个定位参数，所以我们可以只提供日志名称而不需要输入参数名**-LogName**。并且因为这个参数出现在帮助文档的第一个位置，所以我们知道这个日志名称是我们必须提供的第一个参数。

第二个参数：**-InstanceId**。它是可选的，因为它的参数名和参数值放在同一个方括号内。在方括号内，**-InstanceId**本身又处在一个方括号里，意味着它同时还是一个定位参数。它出现在第二个位置，所以我们省略这个参数名，就必须在这个位置提供一个参数值。

参数**-Before**（出现在语法的后面，通过运行**Help Get-EventLog**命令自行查找）是一个可选参数，因为参数名和参数值同在一个方括号里面。**-Before**参数名没有单独放在方括号里，这告诉我们，当选择使用这个参数时，必须输入这个参数名（或者最少是它的别名）。

使用定位参数时的几个技巧。

- 定位参数可以同时出现指定和不指定参数名的情况，但是定位参数必须处在正确的位置。例如，**Get-EventLog -newest 20 -Log Application**是正确的；**System**会被匹配到**-LogName**参数，因为这是第一个位置的参数值，**20**将表示**-Newest**参数值，因为你已经使用了参数名。
- 指定参数名总是正确的。当你这样做了，输入的顺序就变得不重要。**Get-EventLog -newest 20 -Log Application**是正确的，因为我们已经使用了参数名（在这个示例中是**-LogName**，我们这里使用了缩写**-Log**）。
- 如果使用多个定位参数，不要忘了它们的位置。**Get-EventLog Application 0**是可以运行的，**Application**会附加到**-LogName**参数，**0**会附加到**-InstanceId**参数。**Get-EventLog 0 Application**会运行失败，因为**0**会附加**-LogName**参数名，但是却找不到名为“0”的日志。

我们将提供一个最佳实践：一直使用参数名，直到你能顺手地使用每个**Cmdlet**并厌倦了一遍一遍输入常用的参数。在此之后，使用定位参数来节省时间。当需要把一个命令以文件的形式存储在文本文件以方便重用时，通常使用完整的**Cmdlet**名和完整的参数名。这样做的目的是将来可以方便阅读和理解，因为你不需要重复输入参数名（这

毕竟也是你把命令存储在一个文件的目的），这不会增加你太多额外的输入。

在详细的帮助文档中找到定位参数

我们说通常有两种方式来定位参数。第二种方式需要你使用**Help**命令指定**-full**参数来打开帮助文档。

动手实验：运行**Help Get-EventLog-full**命令。记得使用空格一页一页查看帮助文档，如果你想停止查看，可以使用**Ctrl-C**组合键到达帮助文件的末尾。现在，可以通过滚动窗口重复查看整个页面。

一页一页查看，直到你看到类似下面关于**-LogName**参数的信息。

```
-LogName <string>
    指定事件日志。输入一个事件日志的日志名称（Log 属性的值；而非
LogDisplayName）。
    不允许使用通配符。此参数是必需的。

    是否必需?                True
    位置?                    1
    默认值
    是否接受管道输入?        False
    是否接受通配符?          False
```

在前面的例子中，你可以看到这是一个强制参数，并且它是一个定位参数，同时，它出现在**Cmdlet**命令之后的第一个位置。

当学生开始使用一个**Cmdlet**命令的时候，我们总是鼓励他们把焦点放在阅读帮助上，而不只是缩写语法的提示上。阅读帮助可以让我们理解得更加详细，包含参数的使用描述。你可以看到这个参数不接受通配符，这意味着你不能提供类似**App***的参数值，你需要输入日志名的全称，如**Application**。

3.5.4 参数值

帮助文档同样给你提供了每个参数的数据类型。有些参数被称为开关参数，无需任何输入值。在概要语法中，它们看起来如下所示。

```
[-AsString]
```

在详细语法中，它们看起来如下所示。

```
-AsString [<SwitchParameter>]  
  以字符串而非对象的形式返回输出。  
  是否必需?                False  
  位置?                    named  
  默认值  
  是否接受管道输入?        False  
  是否接受通配符?          False
```

通过[<SwitchParameter>]可以确认这是一个开关参数，并不需要任何输入值。开关参数的位置可以随意放置，你必须输入参数名(或者至少是一个缩写)。开关参数总是可选的，这可以让你选择是否使用它们。

其他参数希望获得的数据类型，通常会跟在参数名之后，并使用空格与参数名分开（不是冒号、等号或者其他字符）。在概要语法里面，输入的类型使用尖括号来表明，例如< >:

```
[-LogName] <string>
```

在详细语法中以相同的方式显示:

```
-Message <string>
```

获取其消息中具有指定字符串的事件。可以使用此属性来搜索包含特定单词或短语的消息。允许使用通配符。

是否必需?	False
位置?	named
默认值	
是否接受管道输入?	False
是否接受通配符?	True

下面来看看通常的输入类型。

- **String**——一系列字母和数字，有些时候也会包含空格符。如果出现空格符，那么全部字符串必须包含在引号内。例如，类似 **C:\Windows** 的字符串不需要使用引号，但是 **C:\Program Files** 这样的字符串就需要，因为它包含了一个空格。现在，你可以交替使用单引号或者双引号，但是最好坚持使用单引号。
- **Int, Int32, or Int64**——一个整数类型（整个数字不包含小数）。
- **DateTime**——通常，基于你本地计算机的时区配置，字符串被解释成的日期会有所不同。在美国，通过的日期格式为 **10-10-2010**，即月-日-年。

关于更多类型，我们将在遇到的时候再做讨论。

你也许注意到有些值包含多个方括号：

```
[-ComputerName <string[]>]
```

string后面的方括号并不意味着某些东西是可选的。事实上，**string[]**意味着这个参数可以接受数组、集合，或者是一个列表类型的字符串。在这种情况下，只提供一个值也是符合语法的。

```
Get-EventLog Security -computer Server-R2
```

但是指定多个值也是符合语法的。一个简单的方式是提供一个以逗号为分隔符的列表。**PowerShell**把以逗号为分隔符的列表作为数组值来对待。


```
Get-EventLog Security -computer Server-R2, DC4, Files02
```

再次说明，任何一个单一值中如果包含了空格，就必须使用引号。但是作为一个整体的列表，是不需要使用引号的，只有单一值才需要使用引号。这一点非常重要。下面的命令是符合语法的。

```
Get-EventLog Security -computer 'Server-R2', 'Files02'
```

如果你想为每个值都加上引号也是可以的（即使这些值都需要引号）。但是下面将会出错：

```
Get-EventLog Security -computer 'Server-R2, Files01'
```

在这个示例中，**Cmdlet**命令会查找一个名为**Server-R2, Files01**的计算机。这也许不是你想要的。

另外一种提供列表值的方式是把它们输入到一个文本文件中，每一个值一行。例如：

```
Server-R2  
Files02  
Files03  
DC04  
DC03
```

接着，你可以使用**Get-Content**这个**Cmdlet**来读取这个文件的内容，并且发送这些内容到**-computerName**参数中。你可以强制**Shell**先执行**Get-Content**命令，这样就可以把结果送到这个参数了。

记得高中数学中像 $()$ 的括号可以用来在数学表达式中指定操作的顺序。这同样适用于 **PowerShell**。使用圆括号把命令括起来，就强制这些命令先执行。

```
Get-EventLog Application -computer (Get-Content names.txt)
```

前面一个示例展示了一个有用的技巧：我们可以把**Web**服务器、域名控制器和数据库服务器等不同类型的服务器放到一个文本文件中，接着使用这个技巧再次运行这个包含全部计算机集合的命令。

你也可以使用其他方式来输入一个列表值，包含从活动目录中读取计算机名称。这些技术会更加复杂。在学习一些**Cmdlet**命令之后，你需要学会这些技巧。我们会在后面的章节学习到。

另一种为参数（假设它是一个强制参数）指定多个值的方式是不指定参数。与所有强制参数一样，**PowerShell**将提示你输入参数值。对于接受多个值的参数，你可以输入第一个值并按回车键，继续输入直到完成，最后空白处按回车键，这将告诉**PowerShell**你已经完成输入了。像通常一样，如果你不想被提示输入项，可以按**Ctrl+C**组合键来终止命令。

3.5.5 发现命令示例

我们倾向于通过示例来学习，这就是在本书放置大量示例的原因。**PowerShell**的设计者知道大部分管理员都喜欢示例，这也是他们把大量的示例放置到帮助文档的原因。如果你滚动到**Get-EventLog**帮助文档的末尾，几乎可以发现一打使用这个**Cmdlet**命令的例子。

如果你只想看到示例，我们有一个简单获取到这些示例的方法：在**Help**命令中加入**-example**参数，而不是使用**-full**参数。

```
Help Get-EventLog -example
```

动手实验： 使用这个新的参数来获取一个Cmdlet 命令的示例。

我们喜欢这些示例，尽管它们有些会比较复杂。如果遇到一个对你来说太复杂的示例，请忽略它，并测试其他示例。或者通过一点点的尝试（必须在非生产机器上测试），看你是否知道这个示例是用来干什么的，为什么要这样用。

3.6 访问“关于”主题

在本章的前面部分，我们提到PowerShell的帮助系统包含许多背景主题，可以用来帮助定位指定的Cmdlet命令。这些背景主题通常被称为“关于”主题，因为它们都是以“about_”开头的。你可能还记得在本章的前面，所有的Cmdlet命令都提供一个通用参数集。怎样才可以了解更多关于这些常见的参数？

动手实验： 在你继续读本书之前，确认你是否可以通过帮助系统列出公用参数。

我们将先使用通配符。因为“common”在本书已经被多次使用，所以先从下面的关键字开始。

```
Help *common*
```

这真是一个好的关键字。事实上，这只会帮助主题中匹配到一条记录：About_common_parameters。这个主题将会自动显示，因为只有唯一一条配置的主题。浏览显示的帮助主题，你会发现如下8个通用参数。

```
-Verbose
-Debug
-WarningAction
-WarningVariable
-ErrorAction
-ErrorVariable
-OutVariable
```

-OutBuffer

这个帮助文档提到两个额外的“风险缓解”参数，但是并不是每个Cmdlet命令都提供这两个参数。

在帮助系统中，“关于”这个主题是非常重要的。但是，因为它们没有关联到某个特定的Cmdlet命令，所以很容易被人忽略。如果你运行`help about*`列出所有信息，你也许会吃惊怎么有那么多额外的文档信息隐藏在Shell里面。

表3.1列出了几个第三方脚本和应用程序，可以使PowerShell的帮助更容易访问。

表3.1 PowerShell帮助的第三方脚本和应用程序

资源	URL
一个关于能以图形方式来浏览列出所有可用帮助主题的PowerShell脚本	http://mng.bz/5w8E
一个专用于列出所有可用帮助主题的Windows应用程序	http://www.sapien.com/downloads 登录（可以免费注册）并搜索关键字“Free Tools”
一个可以下载的Windows帮助文档，包含帮助（当然也包括“关于”主题）和PowerShell	http://download.microsoft.com (使用搜索)

3.7 访问在线帮助

PowerShell的帮助文档是由人编写的，这意味着它们并不一定准确无误。除了更新帮助文档（你可以运行`Update-Help`），微软也在其网站上发布帮助文档。PowerShell `help` 命令的`-online`参数，使用它可以在网络中找到你所想要命令的帮助信息：

```
Help Get-EventLog -online
```

微软的TechNet 站点解析这个帮助，并且它通常比安装PowerShell中帮助文档要更新。如果你认为在示例或者语法中发现了一个错误，那就尝试查看在线版本的帮助文档吧。不是所有的Cmdlet全集都包含于PowerShell在线文档，而是由各个产品团队负责（如Exchange团队、SQL Server团队、SharePoint团队等）共同提供帮助文档的更新。但PowerShell在线文档在可用的情况下，会是个不错的文档手册。

我们喜欢在线帮助文档，是因为当我们在PowerShell输入脚本的时候，可以在另一个窗口上阅读文档（帮助文档在Web浏览器也能有良好的格式）。Don通过一个简单的设置就可以使用双屏显示，效果更佳。

3.8 动手实验

注意： 在本实验中，你需要在计算机中运行PowerShell v3或更高版本。我们希望这一章已经传达了掌握PowerShell的帮助系统的重要性。现在是时候通过完成以下任务来磨练你的技能。记住，例子的答案可以在MoreLunches.com上找到。查看这些任务中的斜体字，并使用它们作为线索来完成这一任务。

1. 运行Update-Help并确保它执行无误。这会让你的本机下载一份帮助文档。条件是你的电脑能连上互联网，并且需要在更高特权下运行Shell（这意味着必须在PowerShell的标题中出现“管理员”的字眼）。

2. 哪一个Cmdlet命令能够把其他Cmdlet命令输出的内容转换到HTML？

3. 哪一个Cmdlet命令可以重定向输出到一个文件（*file*）或者到打印机（*printer*）？

4. 哪一个Cmdlet命令可以操作进程（*processes*）？（提示：记住，所有Cmdlet命令包含一个名词。）

5. 你可以用哪一个Cmdlet命令向事务日志 (*log*) 写入 (*write*) 数据?

6. 你必须知道别名是Cmdlet命令的昵称。哪一个Cmdlet可以用于创建、修改或者导入别名 (*aliases*) ?

7. 怎么保证你在Shell中的输入都在一个脚本 (*transcript*) 中, 怎么保存这个脚本到一个文本文件中?

8. 从安全事件 (*event*) 日志检索所有的条目可能需要很长时间, 你怎么只获取最近的100条记录呢?

9. 是否有办法可以获取一个远程计算机上安装的服务 (*services*) 列表?

10. 是否有办法可以看到一个远程计算机运行了什么进程 (*processes*) ?

11. 尝试查看Out-File这个Cmdlet命令的帮助文档。通过这个Cmdlet命令输出到文件每一行记录的默认宽度大小为多少个字符? 是否有一个参数可以让你修改这个宽度?

12. 在默认情况下, Out-File将覆盖任何已经存在具有相同的文件名。是否有一个参数可以预防Cmdlet命令覆盖现有的文件?

13. 你怎么查看在PowerShell中预先定义所有别名 (*aliases*) 的列表?

14. 怎么使用别名和缩写的参数名称来写一条最短的命令, 就能检索出一台名为Server1计算机中正在运行的进程列表?

15. 有多少Cmdlet命令可以处理普通对象? (提示: 记得使用类似“object”的单数名词好过使用类似“objects”的复数名词。)

16. 这一章简单提到了数组 (*arrays*) 。什么帮助主题可以告诉你关于它们的更多信息?

第4章 运行命令

当开始在互联网上查看PowerShell示例时，很容易觉得PowerShell是某种基于.NET Framework的脚本或编程语言。我们的伙伴微软最有价值专家（MVP）奖项获得者，以及大量其他PowerShell用户都是一本正经的极客（Geek）。我们乐于深入挖掘Shell的潜力并发挥它的最大价值。但几乎我们所有人都是以本章标题那样开始：运行命令。这也是本章我们将要做的：没有脚本、没有编程语言，仅仅是运行命令和命令行工具。

4.1 无需脚本，仅仅是运行命令

PowerShell，如其名称所示，是一个Shell。它和你之前可能使用过的Cmd.exe命令行Shell类似，甚至更像是与20世纪80年代第一台PC机一起发布的MS-DOS。它与Unix的Shell也十分类似，比如说20世纪80年代后期的Bash，甚至是20世纪70年代面世最原始的Unix Bourne Shell。虽然PowerShell更加现代，但最终，PowerShell并不是一个类似VBScript或KiXtart的脚本语言。

这些语言和大多数编程语言一样，你在文本编辑器（即使是Windows记事本）中键入大量关键字形成脚本。当脚本完成保存为文件后，可能还需要双击该文件进行测试。PowerShell能够以这种方式工作，但这并不是PowerShell的主要工作模式，尤其是当你开始学习PowerShell时。使用PowerShell，你输入一个命令，然后通过添加一些参数来定制化命令行为，点击返回，立刻就能看到结果。

最终，你会厌倦一遍遍输入同样的命令（和参数），然后你会将其复制粘贴到一个文本文件中，并将文件的扩展名更名为.PS1，然后你瞬间就拥有了一个“PowerShell脚本”。现在，你不再需要一遍遍输入命令，而是直接执行该文件中的脚本。这也和你在Cmd.exe Shell中使用的批处理文件是同一种模式，但相较于脚本或编程而言却要简单许多。

别理解错我们的意思：你可以将PowerShell用得极其复杂。实际上，PowerShell支持与VBScript和其他脚本或编程语言同一种使用模

式。PowerShell拥有能够访问整个.Net Framework底层的能力，我们也看到PowerShell“脚本”实际上与通过Visual Studio编写的C#语言使用模式也十分类似。PowerShell支持这两种不同的使用模式，是因为其设计目标是为了拥有更广阔的使用场景。关键是，不能仅仅是PowerShell可以实现得非常复杂，就意味着你也必须将PowerShell使用到这种程度，也并不意味着你不能以更简单的方式实现非常高效的结果。

来看这样一个类比：你或许有一辆车，如果你和我们一样，或许换机油是你对车做过最复杂的机械性工作。我们并不是汽车专家，也不能重建一个引擎。我们也不能完成如你在电影中所见非常酷的漂移。你从未见过我们在汽车广告中的封闭赛道开车，虽然Jeff的梦想就是这么做（他看了太多的Top Gear（译者注：一档由英国BBC电台出品的汽车节目））。虽然我们并不是专业的赛车手，但是并不会阻挡我们在日常中以更低的复杂度高效驾驶。如果某天我们决定来一场特技驾驶（我们的保险公司会被吓死的），这时或许多学一点汽车工作的原理并掌握一些新的技巧才更有帮助。留给我们进阶的选择一直就在那儿。但目前为止，我们对完成普通驾驶就非常满意。

目前为止，我们依然是一位普通的“PowerShell驾驶员”，以比较简单的方式操纵Shell。无论你是否相信，在该阶段的用户才是PowerShell的主要目标用户。你会发现，在这个阶段，你就能够完成很多难以置信的工作。你仅需要掌握如何在Shell中运行命令即可。

4.2 剖析一个命令

图4.1展示了对复杂PowerShell命令的一个基本剖析。我们称之为一个命令的完整语法形式。我们尝试使用一个有点复杂的命令，这样你就能看到可能出现的所有部分。

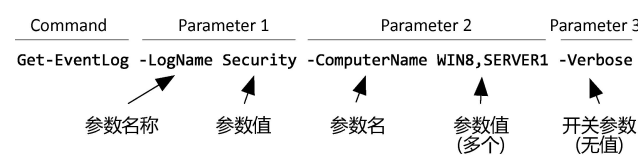


图4.1 剖析一个PowerShell命令

为了确保你能够完全熟悉PowerShell的规则，下面更详细地阐述上一张图中每一部分。

- 名称为Get-EventLog的Cmdlet。PowerShell Cmdlet总是以这种动词-名词形式命名。我们将在下一章关于Cmdlet的章节进一步解释。
- 第一个参数名称为-LogName，并赋值为Security。由于参数值中并不包含任何空格或标点符号，因此并不需要用引号括起来。
- 第二个参数名称为-ComputerName，以逗号分隔列表的形式赋了两个值：Win8和Server1。由于这两个参数中都不包含空格或标点符号，因此这两个参数都不需要用引号括起来。
- 最后一个参数是-Verbose，是一个开关参数。这意味着该参数无须赋值，仅仅指定参数即可。
- 注意：在命令名称和第一个参数之间必须有空格。
- 参数名总是以英文短横线 (-) 开头。
- 参数名之间必须有空格，多个参数值之间也必须有空格。
- 无论参数名之前的破折号，还是参数值本身包含的破折号，都不需要加空格。
- PowerShell不区分大小写。

请逐渐习惯这些规则，并开始对这种精确优雅的输入方式敏感。多注意空格、破折号和其他部分可以最大程度减少PowerShell报低级错误的机会。

4.3 Cmdlet命名惯例

首先，让我们以讨论一些术语开始。据目前我们所知，下面的术语仅仅用于PowerShell。但为了确保对属于理解的统一性，我们还需要详细解释一下：

- **Cmdlet**是一个原生的PowerShell命令行工具。该术语仅仅存在于PowerShell和类似C#的.Net Framework语言中。**Cmdlet**仅仅出现在PowerShell中，所以当你在Google或Bing搜索该关键字时，返回结果主要是关于PowerShell的。该术语读音为“command-let”。
- 函数和**Cmdlet**类似，但不是以.Net语言编写，而是以PowerShell自己的脚本语言编写。
- 工作流是嵌入PowerShell的工作流执行系统的一类特殊函数。
- 应用程序是任意类型的外部可执行程序，包括类似PING、Ipconfig等命令行工具。
- 命令是一个通用的术语，用于代表任何或所有上面提到的术语。

微软已经为**Cmdlet**建了一个命名惯例。因此同样的命名管理也应该被用于函数和工作流。虽然微软并没有强制要求，但开发人员应该遵循该惯例。

规则应该以标准的动词开始，比如**Get**、**Set**、**New**或**Pause**。你可以运行**Get-Verb**查看允许使用的动词列表（虽然只有少部分是常用的，但你大概会看到100个左右）。在动词之后紧接着一个破折号，然后是一个单数形式的名词，比如说**Service**或**Process**或**EventLog**。由于PowerShell允许开发人员自己命名名词，因此并没有一个“**Get-Noun**”的**Cmdlet**来显示所有名词。

这个规则的妙处在哪里？假设我们告诉你如下几个**Cmdlet**名称：**New-Service**、**Get-Service**、**Get-Process**、**Set-Service**等。你能够猜出哪一个命令可以创建一个新的**Exchange**邮箱？你是否能够猜出哪一个命令可以修改活动目录用户？如果你猜是“**Get-Mailbox**”，那么第一个就猜对了。如果你猜“**Set-User**”，那么第二个就非常接近了。实际上是**Set-ADUser**，你可以在活动目录模块的域控制器中找到该用户。重点是，通过一致的命名规则以及有限的动词集合，猜测命令名称变为可能，在此之后才是使用帮助或“**Get-Command**”加通配符验证猜想。估计你所需要的命令名称会变得更加简单，而无须每次都去搜索Google或Bing。

注意：

并不是所有所谓的动词都是动词。虽然微软官方使用术语“动词-名词命名规范”，你仍然能看到类似New、Where等“动词”，请逐渐习惯吧。

4.4 别名：命令的昵称

虽然PowerShell命令名称足够好，并具有良好的 consistency，但仍然可能很长。类似Set-WinDefaultInputMethodOverride的命令名称，即使有Tab键补全，对于输入来说也是太长。虽然命令名称非常清晰——看到名称就能大概猜到其功能，但对于输入来说还是太长。

这也是为什么需要PowerShell别名。别名仅仅是命令的昵称。厌倦了输入Get-Service？尝试下面的代码：

```
PS C:\> get-alias -Definition "Get-Service"
Capability      Name
-----
Cmdlet         gsv -> Get-Service
```

现在你知道Gsv是Get-Service的别名了。

无论是否使用别名，命令的工作方式不会变。参数还是原来的参数，其他部分也不会有任何改变——仅仅是命令名称变得更短。

如果你看到一个别名（网上的一些家伙倾向于使用别名，就好像我们都能够记住所有150个内置别名）而不知道其含义，请查阅帮助：

```
PS C:\> help gsv
NAME
    Get-Service
SYNOPSIS
    Gets the services on a local or remote computer.
SYNTAX
    Get-Service [[-Name] <String[]>] [-ComputerName <String[]>]
    [-DependentServices [<SwitchParameter>]] [-Exclude <String[]>]
    [-Include <String[]>] [-RequiredServices [<SwitchParameter>]]
    [<CommonParameters>]

    Get-Service [-ComputerName <String[]>] [-DependentServices
    [<SwitchParameter>]] [-Exclude <String[]>] [-Include
```

```
<String[]>
    [-RequiredServices [<SwitchParameter>]] -DisplayName
<String[]>
    [<CommonParameters>]

    Get-Service [-ComputerName <String[]>] [-DependentServices
    [<SwitchParameter>]] [-Exclude <String[]>] [-Include
<String[]>]
    [-InputObject <ServiceController[]>] [-RequiredServices
    [<SwitchParameter>]] [<CommonParameters>]
```

在根据别名查阅帮助时，帮助系统将会显示完整命令的帮助，其中也会包含命令的完整名称。

补充说明

你可以使用**New-Alias**创建自定义别名，使用**Export-Alias**导出别名列表。当创建一个别名时，其生命周期只能持续到当前的Shell会话结束。一旦关闭窗口，别名就会不复存在。这也是你需要导出别名的原因，以便后续重新导入。我们通常会避免创建和使用自定义别名，因为这些别名除我们之外的别人无法使用。如果某个用户无法查到**xtd**的含义，这会导致混淆。**xtd**仅仅是我们编造的一个假的别名，不会做任何工作。

4.5 使用快捷方式

这也是PowerShell奇妙的地方。之前提到过PowerShell唯一的方式就是我们之前给你展示的那样，但实际上我们撒谎了。如果你希望在网络上偷取（或者再利用）其他人的示例代码，那首先需要懂得如何看懂它。

除了作为快捷方式的命令的别名之外，参数也同样可以使用别名。总共有三种方式可以实现这一点，每一种都可能造成混淆。

4.5.1 简化参数名称

PowerShell并不强制要求输入完整的参数名称。例如，你可以通过输入**-comp**代替**-ComputerName**，简化的规则是必须输入足够的字母让PowerShell可以识别不同参数。如果既存在**-composite**参数，也存在-

computerName以及-common参数，你至少要输入-compu、-commo和-compo。这是由于上述值是唯一识别参数所需要输入的最少部分。

如果你很希望使用简便方式，那上面就是一个不错的选择。如果你在输入最少部分的参数之后记得按Tab键，PowerShell会帮你自动完成余下的输入。

4.5.2 参数名称别名

尽管参数的别名不在帮助文件或任何方便查阅的地方而难以识别，但参数也拥有别名。比如说，Get-EventLog命令有-ComputerName参数。可以运行下述命令，查阅该参数别名。

```
PS C:\> (get-command get-eventlog | select -ExpandProperty  
parameters). Comp  
utername.aliases
```

上述命令已经用粗体标出命令和参数名称。你可以用任意你希望了解的命令和参数名称进行替换。在本例中，数据结果展示了-Cn是-computerName 的别名，所以你可以运行下述命令：

```
PS C:\> Get-EventLog -LogName Security -Cn SERVER2 -Newest 10
```

Tab键补全将会展示出-Cn这个别名。如果你输入Get-EventLog -C并开始按Tab键，该别名将会出现。但是命令的帮助并不会显示关于-Cn的任何信息，且Tab键补全并不会显示-Cn和-ComputerName实际上hi同一个命令。

4.5.3 定位参数

当你在帮助文件中查看命令语法时，你可以很容易认出定位参数：

```
SYNTAX
```

```

    Get-ChildItem [[-Path] <String[]>] [[-Filter] <String>] [-
Exclude
    <String[]>] [-Force [<SwitchParameter>]] [-Include <String[]>]
[-Name
    [<SwitchParameter>]] [-Recurse [<SwitchParameter>]] [-
UseTransaction
    [<SwitchParameter>]] [<CommonParameters>]

```

在上述语法中，**-Path**和**-Filter**参数是定位参数，这是由于参数名称被中括号给括起来。在完整的帮助文档（本例是`help Get-ChildItem-Full`）中会有更清晰的解释，如下：

```

-Path <String[]>
    Specifies a path to one or more locations. Wildcards are
    permitted. The default location is the current directory (.).

    Required?                false
    Position?                1
    Default value             Current directory
    Accept pipeline input?    true (ByValue, ByPropertyName)
    Accept wildcard characters? True

```

上述帮助明显解释了**-Path**参数在位置1。对于定位参数来说，你无须输入参数名称——仅需要在正确的位置提供参数值，例如：

```

PS C:\> Get-ChildItem c:\users
    Directory: C:\users
Mode                LastWriteTime         Length Name
-----
d----          3/27/2012  11:20 AM                donjones
d-r--          2/18/2012   2:06 AM                Public

```

和下述命令完全相同：

```

PS C:\> Get-ChildItem -path c:\users
    Directory: C:\users
Mode                LastWriteTime         Length Name
-----
d----          3/27/2012  11:20 AM                donjones

```

定位参数的一个弊端是你必须记住每一个位置所代表的参数。你还必须首先按照正确的顺序输入定位参数，然后才能输入命名（非定位）参数。如果你将定位参数的顺序搞混，命令则会失败。对于你可能已经使用多年的简单**DIR**命令来说，如果提供**-Path**参数将会变得很奇怪，没有人会这么做。但对于更复杂的命令来说，比如一行包含3至4个定位参数的命令，将难以记住每一个位置所代表的参数。

比如说，下面命令将会难以阅读和理解：

```
PS C:\> move file.txt users\donjones\
```

下面的版本显式指定了参数名称，将会更容易理解：

```
PS C:\> move -Path c:\file.txt -Destination \users\donjones\
```

下面的版本将参数调换顺序，只有在指定参数名称时才允许这么做：

```
PS C:\> move -Destination \users\donjones\ -Path c:\file.txt
```

我们倾向于不推荐使用定位（也就是不指定参数名）参数，除非你仅仅是即时输入一个命令并不会带来任何后续影响。任何将命令长期保存的方式，包括在批处理文件中或是写入博客中，都要把所有的参数名称带上。我们在本书中尽量不使用不指定参数名称的方式，只有在少数示例中由于命令过长影响到排版时，我们才会使用。

4.6 小小作弊一下：Show-Command

尽管我们拥有多年使用PowerShell的经验，但命令语法的复杂度有时依然会让我们抓狂。PowerShell v3提供的一个非常棒的特性是

Show-Command cmdlet。如果你在命令语法方面遇到困难，包括空格、破折号、逗号、引号或是其他方面，**Show-Command**将成为你的助手。该命令允许你指定你无法用对的命令名称，并以图形化的方式将命令的参数名称展示出来。如图4.2所示，**Tab**键补全不会跨越参数集（在上一章学到的），因此不同参数集之间的参数不会搞混——使用**Tab**键补全并坚持使用。

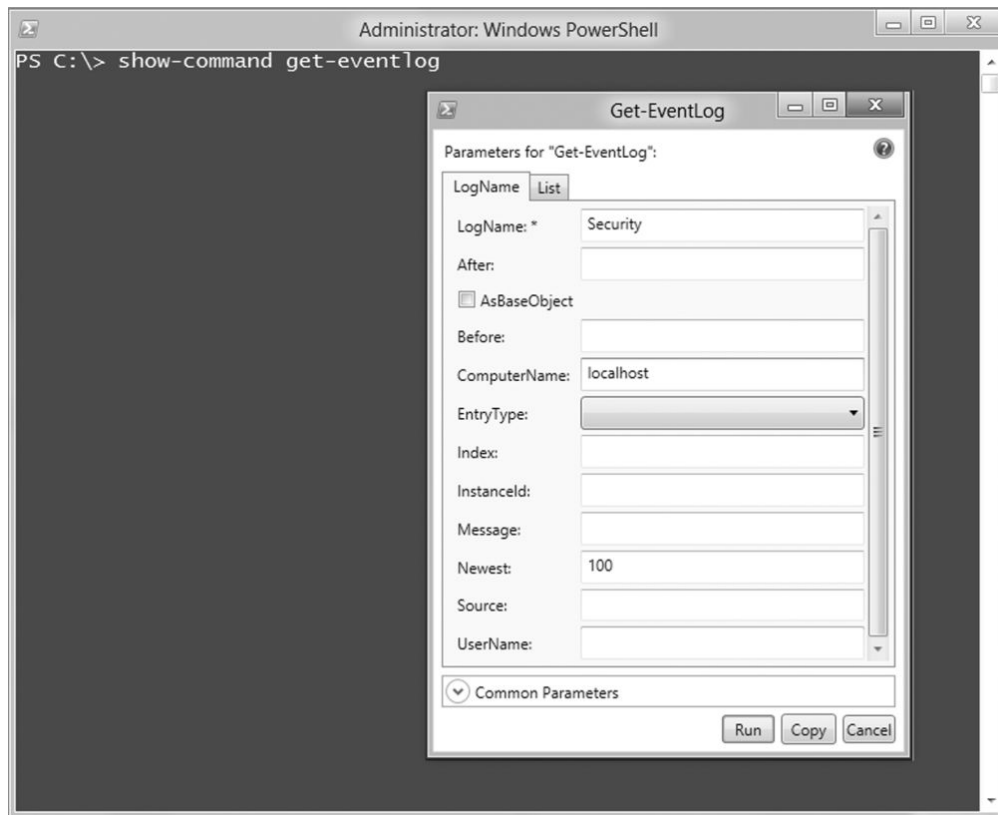


图4.2 Show-Command命令使用图形提示帮助填写命令参数

当完成后，你可以单击运行来执行命令，或使用我更喜欢的选项——单击复制将完成后的命令复制到剪贴板，返回Shell，将命令剪贴（右击控制台，或是在ISE中使用Ctrl+V组合键）到Shell中进行查看。这也是自学PowerShell语法最好的方式，如图4.3所示。你每次都能获得正确的语法。

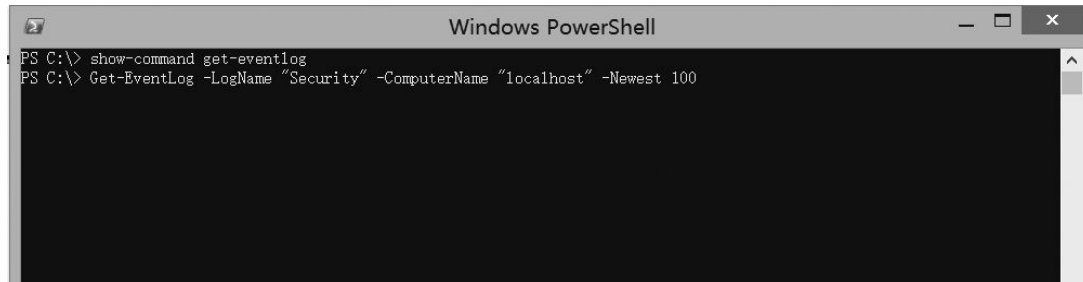


图4.3 基于初始条目对话框，Show-Command生成合适的命令行语法

以这种方式产生的命令，总会是命令的完整形式。完整的命令名称，完整的参数名称，所有的参数名称都显式输入（即，不会出现定位参数）。因此，这种方式可以看到使用PowerShell最完美、推荐并符合最佳实践的方式。

不幸的是，Show-Command一次只能展示一个命令。因此，当希望了解多个命令时，只能逐个使用该命令。

4.7 对扩展命令的支持

目前为止，你所有在Shell中运行的命令（至少是我们建议你运行的命令）都是内置Cmdlet。大约400个Cmdlet都被集成到最新版本的Windows客户端操作系统中，上千个被集成到Windows服务器版本的操作系统中，并且你还能添加更多——类似Exchange Server、Sharepoint Server和SQL Server都包含数以百计的额外Cmdlet。

但是你并不会被局限在仅仅使用随PowerShell一同发行的Cmdlet——你还可以使用一些或许你已经使用多年的外置命令行工具，包括Ping、Nslookup、Ipconfig、Net等。由于这些都不是原生PowerShell Cmdlet，因此你可以按照原来使用这些命令的方法继续使用这些命令。PowerShell将会在后台启动Cmd.exe。由于PowerShell知道如何运行扩展命令，因此返回的结果都会被显示在PowerShell窗口。请尝试运行一些你已经熟悉的CMD命令。我们经常会被问到如何使用PowerShell关联一个普通的网络驱动器——你可以在对象资源管理器中看到那个。我们经常使用的Net Use命令在PowerShell中也能正常工作。

动手实验： 在PowerShell中运行一些之前你熟知的外部命令行工具。是否能够正常工作？哪些命令执行失败？

Net Use示例传递出一个重要信息：使用PowerShell，微软并不是说“你必须重新来过，重新学习所有的一切”，而是说“如果你已经知道了如何完成工作，请继续保持。我们会提供更好、更完整的工具帮助你，但你之前所学依然可用”。PowerShell中不存在“Map-Drive”命令的一个原因是Net Use已经可以很好地完成工作，那为什么不继续使用该命令呢？

注意： 我们已经使用Net Use多年，甚至是PowerShell v1发行之之前，该命令依然是非常好的命令。但PowerShell v3证实微软开始寻找合适的时机推出PowerShell风格的方式来完成这些传统任务。现在你可以发现New-PSDrive命令多了一个-Persisit参数，该参数决定是否启用文件系统提供程序。这样一来，新的磁盘将会在文件资源管理器中被查看到。

有一些确定的例子说明微软已经为一些已经存在的老的命令提供了一些更好的替代工具。比如说，原生的Test-Connection Cmdlet相比之前的提供了更多选项和更灵活的输出方式。还有外部的Ping命令，如果你知道如何使用Ping命令，它可以解决你的所有需求，请立刻使用它。Ping命令在PowerShell中可以正常工作。

综合上面，我们必须透漏出一个严酷的事实：并不是所有的外部命令都可以流畅地运行在PowerShell中，至少如果你不做一些调整是不行的。这是由于PowerShell解析器——Shell的该部分读取你输入的内容并尝试解析出你希望Shell执行什么——并不是每次都能猜对。有时你输入一个外部命令，就会导致PowerShell产生混乱，输出错误信息，因此命令不会生效。

比如说，当一个外部命令拥有很多参数时，事情就变得很难办。这也是PowerShell在大多数场景下无法工作的情形。我们深入其不能正常工作的细节，但可以提供下面的命令，从而确保运行一个命令且其参数可以准确无误：

```
$exe = "C:\Vmware\vcbMounter.exe"
$host = "server"
$user = "joe"
$password = "password"
$machine = "somepc"
$location = "somelocation"
$backupType = "incremental"
```

```
& $exe -h $host -u $user -p $password -s "name:$machine" -r  
$location -t  
$backupType
```

假设你有一个名为**vcbMounter.exe**的外部命令（这是一个由某些VMWare虚拟化产品所提供的真实命令；如果你从未使用或安装过该命令，没有关系——大多数传统的命令行工具都以同样的方式工作，所以这依然是一个很好的教学案例），该命令接受 6 个参数：

- -h for the host name
- -u for the user name
- -p for the password
- -S for the server name
- -r for a location
- -t for a backup type

我们所做的是将不同的元素——可执行路径和名称，以及所有的参数值放入容器。这部分操作以\$开始。这使得PowerShell将这些值当作一个单元，而不是尝试对其进行解析来发现是否包含命令或特殊字符等。然后我们使用调用操作符，将该值传递给可执行名称，还有所有的参数值。这种方式可以在PowerShell中执行的几乎所有的命令行工具中生效。

听上去很复杂？好吧，我们有一些好消息：在PowerShell第三版中，你不必再如此纠结，仅需要在外部的命令名称之后加两个破折号。如果你这么做，PowerShell甚至不会解析该命令，仅仅是将该命令传递到Cmd.exe中。这意味着你基本上可以使用Cmd.exe的语法在PowerShell中运行任何命令，而不用担心该命令是如何被PowerShell解析的。

4.8 处理错误

在刚开始使用PowerShell时无可避免地会遇见丑陋的红色文本提示，在不同水平阶段依然可以遇到，甚至当你成为专家级的Shell用户时也避免不了。我们都能遇到，但不要让红字把你逼疯。

先不管用于警告目的的红字，PowerShell的错误信息的目的是用于帮助。例如，如图4.4所示，红字尝试展示给你PowerShell错误的地方。

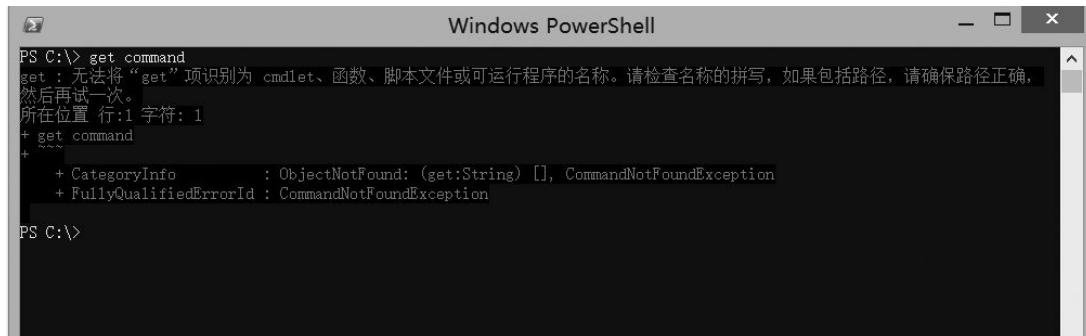


图4.4 解释PowerShell的错误信息

错误信息几乎总是会包括PowerShell认为有歧义地方的行数和字符数。在图4.4中，是第一行，字符1——就是命令开始部分。其表达的意思为“你输入了‘get’，我不知道该词的意思”。这是由于我们输错了命令，正确应该是Get-Command，而不是Get Command。那么图4.5是什么情况？

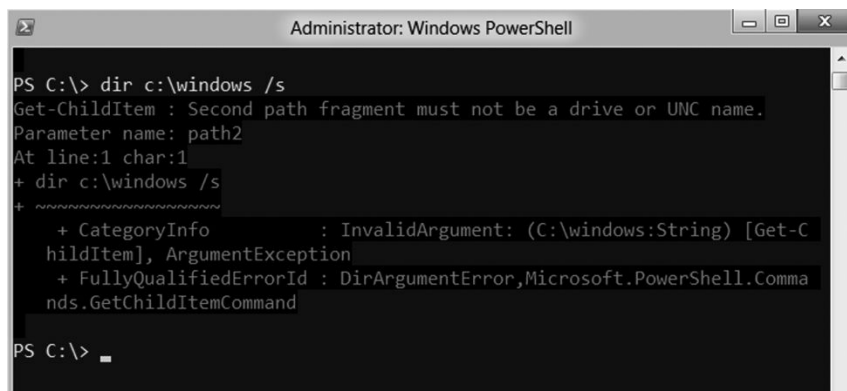


图4.5 “第二路径片段”是什么？

图4.5中所示的错误信息“第二路径不得为驱动器或UNC名称”让人感到困惑，什么第二路径？我们并没有输入第二路径。我们输入了一

个路径c:\windows和一个命令行参数/s，不是吗？

当然不是。解决该类问题最简便的方式就是阅读帮助，并完整输入命令。如果你输入`Get-ChildItem-path C:\Windows`，就会发现/s并不是正确的语法。我们希望该值对应的参数是`-recurse`。有时，错误信息并不一定很有帮助，就好像你和PowerShell说的不是同一种语言。当然，PowerShell不可能改变其语言，那么只能是你错了，所以你得去改变。通过咨询帮助并拼写出完整的命令和参数，通常都是解决问题最快的方式。还有不要忘了使用`Show-Command`来找出正确的语法。

4.9 常见误区

当合适时，我们会在一章中安排一个简短的小节，包含我们在教学过程中存在的一些常见误区。这样做的目的是帮助其他像你一样的管理员，从而避免该类问题——或是至少在你开始使用Shell时对这些问题的解决办法。

4.9.1 输入Cmdlet名称

首先是输入Cmdlet名称。该名称永远是动词-名词形式，比如说`Get-Content`。下面是我看到的一些新手尝试输入的命令，但显然难以奏效：

- `Get Content`
- `GetContent`
- `Get=Content`
- `Get_Content`

其中一些问题是由于输入错误（比如说“=”，而不是“-”），还有一些是省略破折号。我们都会将命令读成“`Get Content`”，省略了破折号。但输入时必须输入破折号。

4.9.2 输入参数

参数同样需要正确书写。参数可以不赋值，比如说`-recurse`，在参数名称之前加上破折号。但必须在`Cmdlet`名称和参数之间加空格，参数之间也需要空格。下述命令都正确：

- `Dir-rec` （可以使用参数名称的简写）
- `New-PSDrive-name DEMO-psprovider FileSystem-root \\Server\Share` 但下述写法不正确：
- `Dir-rec` （在名称和参数之间没有空格）
- `New-PSDrive-nameDEMO` （参数和值之间没有空格）
- `New-PSDrive-name DEMO-psprovider FileSystem` （在第一个参数值和第二个参数名之间没有空格）

PowerShell并不会挑剔大小写问题，也就是说`dir`和`DIR`并无不同。`-RECURSE`、`-recurse`和`-Recurse`也是如此。但PowerShell会挑剔空格和破折号的写法。

4.10 动手实验

注意： 对于本次实验，你需要Windows 8（或更新版本）或Windows 2012（或更新版本）的计算机来运行PowerShell。

请使用在本章以及之前关于帮助系统章节所学的内容，使用PowerShell完成下述任务：

1. 显示正在运行的进程列表。
2. 显示最新的100个应用程序日志（请不要使用`Get-WinEvent`，我们已经为你展示过完成该任务的另一个命令）。
3. 显示所有类型为“`Cmdlet`”的命令（我们已经展示了`Get-Command`，你还需要阅读帮助文档，从而找出缩小该列表范围，正如本次动手实验所要求的）。

4. 显示所有的别名。
5. 创建一个新的别名。使用该别名，你可以运行“D”获取目录列表。
6. 显示以字母M开头的服务名称。同样，你需要阅读帮助文档找出所需的命令。请不要忘了星号（*），这是PowerShell中通用的通配符。
7. 显示所有的Windows防火墙规则，你需要使用Help或Get-Command找出所需的Cmdlet。
8. 显示所有Windows防火墙的入站规则。可以使用和之前任务同样的Cmdlet，但你需要阅读帮助文档找出所需的参数以及可选值。

我们希望上述任务对你来说很直白。如果是这样，那就太好了。你已经利用现有的命令行技巧来使得PowerShell帮助你完成实际的工作。如果你是命令行世界的新手，那么上述任务将会是你学习本书其他章节的敲门砖。

第5章 使用提供程序

PowerShell中较难以理解的一部分是如何使用提供程序。在这里提前声明，本章某些内容或许对你们来说有点难。我们期许读者对Windows的文件系统比较熟悉，比如，你们可能知道如何通过Windows命令行窗口管理文件系统。请记住，我们会利用与命令行管理文件系统类似的方式解释一些内容，读者可以借助之前所熟悉的文件系统的知识作为铺垫，从而更好地使用提供程序。同时，也请谨记，PowerShell并不是Cmd.exe。你可能觉得某些东西看起来差不多，但是我们确信它们大有不同。

5.1 什么是提供程序

一个PowerShell的提供程序，或者说PSProvider，其本质上是一个适配器。它可以访问某些数据存储介质，并使得这些介质看起来像是磁盘驱动器一样。你可以通过下面的命令查看当前Shell中已经存在的提供程序。

```
PS C:\Users\gaizai> Get-PSProvider
Name                               Capabilities                               Drives
-----
Alias                               ShouldProcess                               {Alias}
Environment                         ShouldProcess                               {Env}
FileSystem                         Filter, ShouldProcess, Credentials {C, D, A}
Function                           ShouldProcess                               {Function}
Registry                           ShouldProcess, Transactions               {HKLM, HKCU}
Variable                           ShouldProcess                               {Variable}
```

我们可以通过模块或者管理单元将一些提供程序添加到PowerShell中，这也是PowerShell仅支持的两种扩展方式。（我们会在第7章中讲解该部分知识。）有些时候，如果启用了某些PowerShell功能，可能也会新增一个新的PSProvider。比如，当开启了远程处理时（将在第13章中讨论该话题），会新增一个PSProvider，比如：

```
PS C:\Users\gaizai> Get-PSProvider
Name                               Capabilities                               Drives
-----
Alias                               ShouldProcess                               {Alias}
Environment                         ShouldProcess                               {Env}
FileSystem                         Filter, ShouldProcess, Credentials {C, D, A}
Function                           ShouldProcess                               {Function}
```


Registry	ShouldProcess, Transactions	{HKLM, HKCU}
Variable	ShouldProcess	{Variable}
WSMan	Credentials	{WSMan}

我们可以看出每个提供程序都有各自不同的功能。这非常重要，因为这将决定我们如何使用这些提供程序。下面是常见的一些功能描述。

- **ShouldProcess**——意味着这部分提供程序支持-WhatIf和-Confirm参数，保证我们在正式执行这部分脚本之前可以对它们进行测试。
- **Filter**——意味着在Cmdlet中操作提供程序的数据时，支持-Filter参数。
- **Credentials**——意味着该提供程序允许使用可变更的凭据去连接数据存储。这也就是-Credentials参数的作用。
- **Transactions**——意味着该提供程序支持事务，也就是允许你在该提供程序中将多个变更作为一个原子操作进行提交或者全部回滚。

你也可以使用某个提供程序去创建一个PSDrive。PSDrive可以通过一个特定的提供程序去连接到某些存储数据的介质。这和Windows资源管理器中类似，本质上是创建了一个驱动器映射。但是由于PSDrive使用了提供程序，除了可以连接磁盘之外，还能连接更多的数据存储介质。运行下面的命令，可以看到当前已连接的驱动器。（译者注：返回基于PowerShell 3.0）

```
PS C:\Users\gaizai> Get-PSDrive
```

Name	Used (GB)	Free (GB)	Provider	Root
A			FileSystem	A:\
Alias				Alias
C	29.40	97.60	FileSystem	C:\
Cert				Certificate \
D	1.26	283.74	FileSystem	D:\
Env				Environment
Function				Function
HKCU			Registry	HKEY_CURRENT_USER
HKLM			Registry	HKEY_LOCAL_MACHINE
Variable				Variable
WSMan				WSMan

在上面返回的列表中，可以看到有三个驱动器使用了FileSystem提供程序，两个使用了Registry提供程序，等等。PSProvider会适配对应的数据存储，通过PSDrive机制使得数据存储可被访问，然后可以使用一系列Cmdlets去查阅或者操作每个PSDrive呈现出来的数据。通常我们可以通过下面的命

令来查询某个PSDrive的Cmdlets中有哪些命令的名称中包含“Item”字符（译者注：返回结果基于PowerShell 3.0）。

```
PS C:\Users\gaizai> Get-Command -noun *Item*
CommandType Name      Name
-----
Function     Get-DAEntryPointTableItem  DirectAccessClientComponents
Function     New-DAEntryPointTableItem  DirectAccessClientComponents
Function     Remove-DAEntryPointTableItem
DirectAccessClientComponents
Function     Rename-DAEntryPointTableItem
DirectAccessClientComponents
Function     Reset-DAEntryPointTableItem  DirectAccessClientComponents
Function     Set-DAEntryPointTableItem  DirectAccessClientComponents
Cmdlet       Clear-Item                  Microsoft.PowerShell.Management
Cmdlet       Clear-ItemProperty          Microsoft.PowerShell.Management
Cmdlet       Copy-Item                   Microsoft.PowerShell.Management
Cmdlet       Copy-ItemProperty           Microsoft.PowerShell.Management
Cmdlet       Get-ChildItem               Microsoft.PowerShell.Management
Cmdlet       Get-ControlItem             Microsoft.PowerShell.Management
Cmdlet       Get-Item                    Microsoft.PowerShell.Management
Cmdlet       Get-ItemProperty            Microsoft.PowerShell.Management
Cmdlet       Invoke-Item                 Microsoft.PowerShell.Management
Cmdlet       Move-Item                   Microsoft.PowerShell.Management
Cmdlet       Move-ItemProperty           Microsoft.PowerShell.Management
Cmdlet       New-Item                    Microsoft.PowerShell.Management
Cmdlet       New-ItemProperty            Microsoft.PowerShell.Management
Cmdlet       Remove-Item                 Microsoft.PowerShell.Management
Cmdlet       Remove-ItemProperty         Microsoft.PowerShell.Management
Cmdlet       Rename-Item                 Microsoft.PowerShell.Management
Cmdlet       Rename-ItemProperty         Microsoft.PowerShell.Management
Cmdlet       Set-Item                    Microsoft.PowerShell.Management
Cmdlet       Set-ItemProperty            Microsoft.PowerShell.Management
Cmdlet       Show-ControlItem            Microsoft.PowerShell.Management
```

我们将在系统中使用上述Cmdlet或者它们的别名来调用提供程序。或许对你而言，文件系统应该算是最熟悉的提供程序了，所以我们将会从文件系统PSProvider开始学习。

5.2 FileSystem的结构

Windows 文件系统主要由三种对象组成：磁盘驱动器、文件夹和文件。磁盘驱动器是最上层的对象，包含文件夹和文件。文件夹是一种容器对象，它可以包含文件以及其他文件夹。文件不是一种容器对象，它更可以算作最小单位的对象。

你或许习惯于通过Windows资源管理器来查看Windows的文件系统，如图5.1所示。在图中，我们可以直观地观察到磁盘驱动器、文件夹和文件的层级分布。

PowerShell中的术语和文件系统略有不同。因为PSDrive可能不是指向某个文件系统——比如PSDrive可以映射到注册表（显然注册表并不是一种文件系统），所以PowerShell并不会使用“文件”以及“文件夹”的说法。相反，PowerShell采用更通俗的说法“项”（Item）。一个文件或者一个文件夹都叫作项，尽管本质上是两种不同的项。这也就是为什么前面返回的Cmdlet名字中都有“Item”字符。

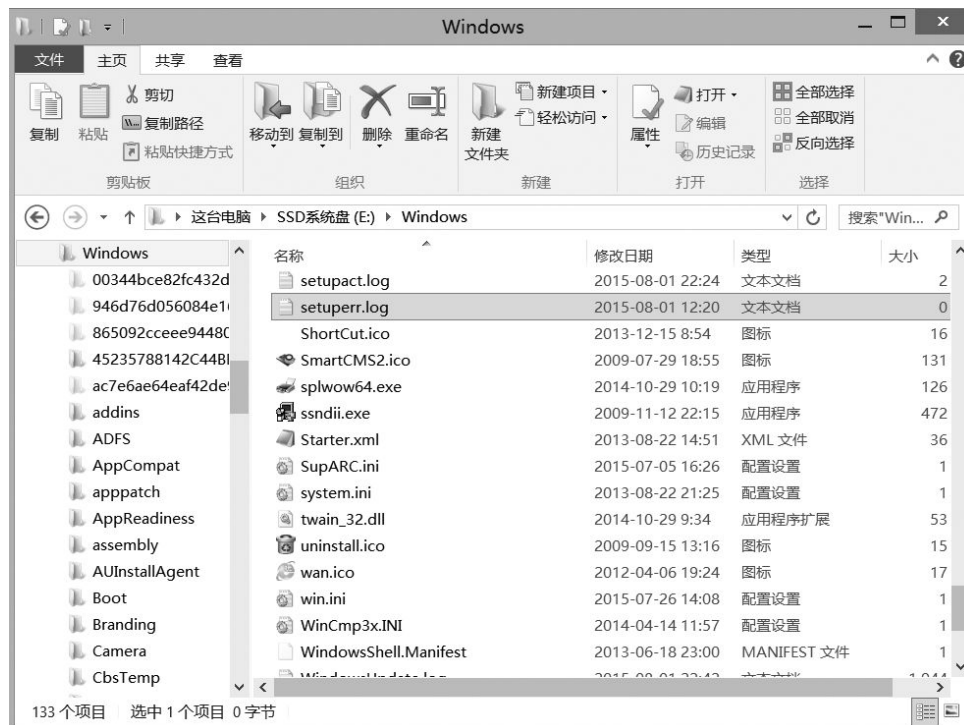


图5.1 在Windows资源管理器中查看文件、文件夹及磁盘

每个项基本上都会存在对应的属性。比如，一个文件项可能有最后写入的时间，是否只读等属性。一些项，比如文件夹，可能包含子项（子项包含在文件夹项中）。了解这些信息会有助于你们理解前面演示的命令列表中的名词以及动词：

- 比如Clear、Copy、Get、Move、New、Remove、Rename以及Set等动词可以应用于这些项（比如文件或者文件夹）以及它们对应的属性（比如该项最后写入的时间或者该项是否只读）。
- 单个对象对应的项名词，比如文件或者文件夹。

- **ItemProperty**代表一个项对应的属性。比如只读、项创建时间、长度等。
- 子项代表一个项（比如文件或者子文件夹）包含于另外一个项（文件夹）中。

需要记住的是，这些**Cmdlet**都是通用的，因为它们需要处理各种不同的数据源。但是某些**Cmdlet**在某些特定场合下不一定能正常工作。比如，**FileSystem**提供程序不支持事务，所以文件系统驱动器下的**Cmdlet**中的命令都不支持**-UseTransaction**参数。再比如，注册表不支持**Filter**功能，所以注册表驱动器下的**Cmdlet**也都不支持**-Filter**参数。

某些**PSProvider**并不具有对应的项属性。比如，**Environment**这个**PSProvider**主要用来构造PowerShell中可用的ENV：类型驱动器（如Env:\PSModulePath）。这个驱动器主要的作用是访问Windows中的环境变量，但是如下所示，它并没有对应的项属性。

```
PS C:\Users\gaizai> Get-ItemProperty -Path Env:\PSModulePath
Get-ItemProperty : 无法使用接口。此提供程序不支持
    IPropertyCmdletProvider 接口。
所在位置 行:1 字符: 17
+ Get-ItemProperty <<<< -Path Env:\PSModulePath
    + CategoryInfo          : NotImplemented: (:) [Get-ItemProperty],
PSNotSupportedException
    +FullyQualifiedErrorId: NotImplemented,
Microsoft.PowerShell.Commands.
    GetItemPropertyCommand
```

对刚接触Windows PowerShell的朋友而言，由于每个**PSProvider**都不尽相同，可能会导致你们无法很好地理解各种提供程序。你们必须去了解每个提供程序能够实现什么功能，并且认识到即便**Cmdlet**知道如何实现某些功能，但是并不意味着该提供程序真正支持对应的操作。

5.3 文件系统——其他数据仓储的模板

其他形式的数据源存储衍生于文件系统，所以严格算起来，文件系统可以算作其他数据仓储的模板。例如，图5.2展示了Windows注册表的结构。

注册表以类似文件系统的结构呈现，其中注册表的键等同于文件系统中的文件夹，对应的键值类似于文件系统中的文件，等等。正是这种广泛的相似性，使得文件系统成为其他形式数据源的最佳模板。所以当用PowerShell访问其他数据存储的时候，显示为驱动器的形式（可以依次展开为项以及查看对应的属性）。但是相似性到这一层级也就结束了：如果你再继续向下展开，那么你会发现不同形式的存储其实差别很大。这也就是为什么各种项的Cmdlet支持如此多的功能，但是并不是每个功能在每种存储中都能运行。

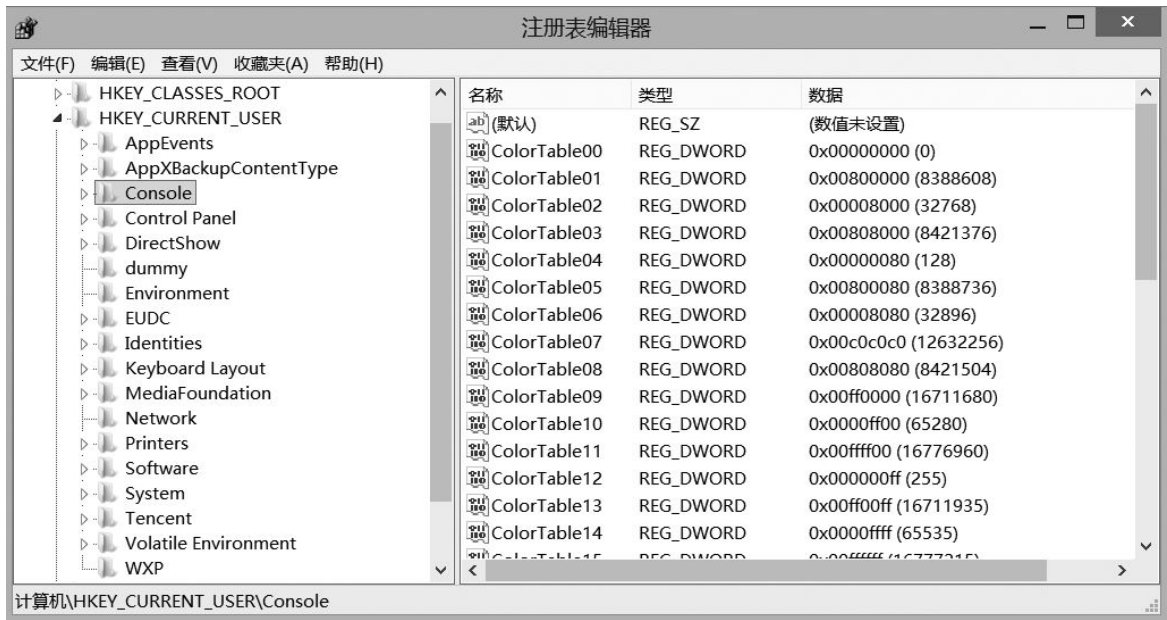


图5.2 注册表和文件系统具有相同的分层结构

5.4 使用文件系统

在使用提供程序时，需要熟悉的另外一个Cmdlet是Set-Location。该参数的功能是将Shell中当前路径变更为不同路径，比如变更到另一个文件夹下：

```
PS C:\Users\gaizai> Set-Location -Path C:\Windows
PS C:\Windows>
```

你可能对该命令的另一种写法cd更为熟悉，其实就是cmd.exe中的change directory的简写。

```
PS C:\Windows> cd 'C:\Program Files'
PS C:\Program Files>
```

这里我们使用了该别名，然后传入特定的路径作为位置参数。

PowerShell中另外一个比较棘手的任务是创建新的项。比如，如何创建一个新的目录。执行**New-Item**，之后会弹出一个新的窗口。

```
PS C:\Users\gaizai> New-Item testFolder
Type:
```

需要注意的是，**New-Item**这个**Cmdlet**在很多地方都是通用的——它根本无法得知你是想新建一个文件夹。这个**Cmdlet**可以用来新建文件夹、文件、注册表项以及其他项，所以你必须告知你希望创建的类型是什么。

```
PS C:\Users\gaizai> New-Item testFolder
Type: Directory

    目录: C:\Users\gaizai
Mode                LastWriteTime         Length Name
-----
d-----          2015/1/5  14:18             testFolder
```

PowerShell中也包含**MKDir**命令。很多人都认为该命令是**New-Item**的别名，但是你们在执行**MKDir**之后，并不需要输入类型。

```
PS C:\Users\gaizai> Mkdir test2

    目录: C:\Users\gaizai
Mode                LastWriteTime         Length Name
-----
d-----          2015/1/5  14:22             test2
```

你可能已经发现，**Mkdir**是一个函数，而并不是一个别名。但实际上，它仍然调用了**New-Item**，只不过隐式赋予了**-Type Directory**这个参数，这样使得**MkDir**看起来更像一种**Cmd.exe**。请记住这一点以及其他的一些小细节，当使用到这些提供程序时，会很有用处。你知道并不是每个提供程

序都是一样，并且项的Cmdlet又是非常通用的，所以在真正使用这些提供程序之前需要思考更多。

5.5 使用通配符以及绝对路径

大部分项的 Cmdlet 都包含了 -Path 属性。默认情况下，该属性支持通配符输入。比如，我们查看 Get-ChildItem 的完整帮助文档，如下所示：

```
PS C:\Users\gaizai> Get-Help Get-ChildItem -Full
-Path
    指定一个或多个位置的路径。允许使用通配符

    默认位置为当前目录 (.)。

    是否必需?          False
    位置?              1
    默认值              Current directory
    是否接受管道输入?   true (ByValue, ByPropertyName)
    是否接受通配符?     True
```

“*”通配符代表0个或者多个字符，“?”通配符仅代表单个字符。你应该曾经多次使用过这两种通配符，当然你可能使用的是Get-ChildItem的别名Dir。

```
PS C:\windows> Dir *.exe

    目录: C:\windows

Mode                LastWriteTime         Length Name
-----
-a---             2012/7/26          3:08         75264 bfsvc.exe
-a---             2013/6/1           11:34       2391280 explorer.exe
-a---             2012/11/6           4:20         883712 HelpPane.exe
-a---             2012/7/26          3:08         17408 hh.exe
-a---             2012/7/26          3:08        159232 regedit.exe
-a---             2012/7/26          3:08        126464 splwow64.exe
-a---             2012/7/26          3:21         10752 winhlp32.exe
-a---             2012/7/26          3:08         10752 write.exe
```

前面例子中列出来的通配符和微软的文件系统中一样（都是采用MS-DOS中的方式）。“*”和“?”比较特殊，它们是通配符，所以在文件或者文件

我们现在先将路径切换到HKEY_CURRENT_USER，在PowerShell中显示为HKCU:驱动器。

```
PS C:\windows> Set-Location -Path HKCU:
```

接下来看注册表的右边。

```
PS HKCU:\> Set-Location -Path Software
PS HKCU:\Software> Get-ChildItem

    Hive: HKEY_CURRENT_USER\Software

Name                                Property
----                                -
AppDataLow
Microsoft
Mine                                (default) : {}
Policies
Wow6432Node
Classes

PS HKCU:\Software> Set-Location Microsoft
PS HKCU:\Software\Microsoft> Get-ChildItem

    Hive: HKEY_CURRENT_USER\Software\Microsoft

Name                                Property
----                                -
Active Setup
Advanced INF Setup
Assistance
Command Processor                  PathCompletionChar : 9
                                   EnableExtensions       : 1
                                   CompletionChar       : 9
                                   DefaultColor        : 0
CTF
EventSystem
Feeds
FTP                                Use PASV : yes
IME
Internet Connection Wizard         Completed : 1
Internet Explorer
MSF
ServerManager                      InitializationComplete      : 1
                                   CheckedUnattendLaunchSetting : 1
SystemCertificates
WAB
Windows
Windows NT
```

测试到这里基本上就结束了（后面的部分基本上都是重复的命令）。你可以看到，我们前面都是用**Cmdlet**的全称，并没有使用它们的别名，这样可以更加强化我们对**Cmdlet**本身的认识。

```
PS HKCU:\Software\Microsoft> Set-Location .\Windows
PS HKCU:\Software\Microsoft\Windows> Get-ChildItem

    Hive: HKEY_CURRENT_USER\Software\Microsoft\Windows

Name                                     Property
----


| Name                    | Property                     |       |
|-------------------------|------------------------------|-------|
| CurrentVersion          | Composition                  | : 1   |
| DWM                     | ColorizationColor            | :     |
| 3226847725              | ColorizationColorBalance     | : 87  |
|                         | ColorizationAfterglow        | :     |
| 3226847725              | ColorizationAfterglowBalance | : 10  |
|                         | ColorizationBlurBalance      | : 3   |
|                         | EnableWindowColorization     | : 1   |
|                         | ColorizationGlassAttribute   | : 1   |
| Roaming                 |                              |       |
| Shell                   |                              |       |
| Windows Error Reporting | Disabled                     | : 0   |
|                         | MaxQueueCount                | : 50  |
|                         | DisableQueue                 | : 0   |
|                         | LoggingDisabled              | : 0   |
|                         | DontSendAdditionalData       | : 0   |
|                         | ForceQueue                   | : 0   |
|                         | DontShowUI                   | : 0   |
|                         | ConfigureArchive             | : 1   |
|                         | MaxArchiveCount              | : 500 |
|                         | DisableArchive               | : 0   |
|                         | LastQueuePesterTime          | :     |
| 130649182874302227      |                              |       |
|                         | LastQueueNoPesterTime        | :     |
| 130649188485744670      |                              |       |


```

你可以在该列表中看到**EnableWindowColorization**的键值，现在将它修改为0。

```
PS HKCU:\Software\Microsoft\Windows> Set-ItemProperty -Path DWM -
PSPropert Enabl
```

```
eWindowColorization -Value 0
```

下面再执行之前的命令来确认修改已经生效。

```
PS HKCU:\Software\Microsoft\Windows> Get-ChildItem

Hive: HKEY_CURRENT_USER\Software\Microsoft\Windows

Name                Property
----                -
CurrentVersion
DWM                  Composition           : 1
                    ColorizationColor       :
3226847725
                    ColorizationColorBalance : 87
                    ColorizationAfterglow     : 3226847725
                    ColorizationAfterglowBalance : 10
                    ColorizationBlurBalance   : 3
                    EnableWindowColorization  : 0
                    ColorizationGlassAttribute : 1
Roaming
Shell
Windows Error Reporting Disabled : 0
                    MaxQueueCount           : 50
                    DisableQueue            : 0
                    LoggingDisabled          : 0
                    DontSendAdditionalData    : 0
                    ForceQueue               : 0
                    DontShowUI               : 0
                    ConfigureArchive         : 1
                    MaxArchiveCount          : 500
                    DisableArchive           : 0
                    LastQueuePesterTime      :
130649182874302227
                    LastQueueNoPesterTime    :
130649188485744670
```

到这里，这个示例已经全部完成，可以看到这个值的修改已经生效。采用这种方法，你可以处理其他提供程序类似的问题。

5.7 动手实验

注意：

本章动手实验环节，需要运行3.0版本或者版本更新的PowerShell。

完成如下任务：

1. 在注册表中，定位到 `HKEY_CURRENT_USER\software\microsoft\Windows\currentversion\explorer`。选中“Advanced”项，然后修改 `DontPrettyPath` 的值为 0。
2. 创建空文件 `C:\Test.txt`（使用 `New-Item` 命令）。
3. 尝试使用 `Set-Item` 去修改 `Test.txt` 的内容为 `TESTING`，是否可行？或者是否有报错？同时，也请想一下：为什么会报错？
4. `Get-ChildItem` 的 `-Filter`、`-Include` 和 `-Exclude` 参数之间有什么不同？

5.8 进一步学习

你可以看到，对大部分的其他软件程序包而言都存在提供程序，比如 **Internet Information Service (IIS)**、**SQL Server**，甚至是活动目录。很多时候，这些产品的开发者都会选择使用提供程式，因为这样他们的产品才会具有动态扩展功能。他们不知道以后还会有什么功能加到他们的产品中，所以他们并不会写一个静态的命令集。提供程序可以保证开发者能一致性地动态扩展他们的结构，所以特别是对 **IIS** 和 **SQL Server** 团队而言，都会搭配使用 `Cmdlet` 和提供程序。

如果你需要使用这些产品（如果是 **IIS**，那么请使用 7.5 或者之后的版本；如果是 **SQL Server**，我们建议使用 **SQL Server 2012** 或者之后的版本），请花费一定的时间去研究一下对应的提供程序。那么你会发现，这些产品研发部门已经将其“驱动器”结构安排得很好，因此你很容易发现如何使用本章中讲解到的 `Cmdlet` 命令去查看以及修改对应的配置选项或者其他详细配置。

第6章 管道：连接命令

在第4章中已经介绍过在PowerShell中运行命令的方式和其他Shell并无不同：输入一个命令名，传输给它一些参数，然后按“Enter”键。让PowerShell独树一帜的不是运行命令的方式，而是它提供了管道功能，通过管道功能，只需要在一个序列行中，多个命令就可以很好地彼此连接。

6.1 一个命令与另外一个命令连接：为你减负

PowerShell通过管道（pipeline）把命令互相连接起来。管道通过传输一个命令，把其输出作为另外一个Cmdlet的输入，使得第二个命令可以通过第一个的结果作为输入并联合起来运行。

你已经见过如“Dir | More”命令的运行情况，它把“Dir”命令的输出以管道方式传输给“More”命令。“More”命令把目录每次展现到一个页中。PowerShell把管道的概念有效延伸。实际上，PowerShell的管道类似Unix和Linux的Shell中的管道功能。你将会在下面认识到，PowerShell的管道功能是非常强大的。

6.2 输出结果到CSV或XML文件

下面尝试几个命令，比如：

- Get-Process （或者Ps）
- Get-Service （或者Gsv）
- Get-EventLog Security-newest 100

这里提到这些命令是因为它们相对简单、直观。其中括号部分是分别对应“Get-Process”和“Get-Service”的别名。对于“Get-EventLog”，我们强制使用了“-newest”参数，避免命令运行太久。

动手实验： 选择你想尝试的命令动手尝试。下面将使用“Get-Process”作为演示。当然，你可以选择其他命令，或者都尝试，以便查看它们的差异。

当运行“Get-Process”时，屏幕会显示出图6.1的结果。

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
161	20	3224	4424	100		2268	AppleMobileDeviceService
85	6	1396	1348	114		5552	appverif
85	6	1396	1352	114		5568	appverif
85	6	1392	1352	114		5584	appverif
84	9	9072	2672	65		5608	appverif
72	6	972	2016	17		1844	AsLdrSrv
155	15	3356	9384	99	0.11	1360	AsusTPCenter
46	6	1156	4440	50	0.02	6676	AsusTPHelper
136	11	1940	8430	82	0.06	9008	AsusTPLoader
65	5	804	840	42		5800	AtBroker
65	5	792	840	42		5816	AtBroker
65	5	808	844	42		5840	AtBroker
65	5	808	852	42		5872	AtBroker
100	13	2356	10176	114	0.13	3640	ATKOSD2
166	14	10304	12696	51	93.45	6360	audiodg
122	10	15628	14924	98		3652	bootim
122	10	15844	15036	98		5216	bootim
122	10	15632	15084	98		5576	bootim
122	10	15716	15060	98		5748	bootim
116	12	4568	4552	78		5208	calc
116	12	4584	4580	78		5624	calc
116	12	4580	4560	78		5704	calc
116	12	4584	4560	78		5808	calc
70	8	1424	5936	74	0.03	10296	caller64
90	7	1248	1456	30		5144	CameraSettingsUIHost
92	7	1260	1464	30		5212	CameraSettingsUIHost
91	7	1328	1532	30		5448	CameraSettingsUIHost
91	7	1248	1468	30		5916	CameraSettingsUIHost
89	6	1160	1388	30		5280	CertEnrollCtrl
90	6	1164	1396	30		5892	CertEnrollCtrl

图6.1 “Get-Process”的输出是一个带有几列信息的表格

虽然屏幕上展示了结果，但是也许不是你想要的，比如如果你想把内存和CPU的利用率整理成一些图表，那么可能需要把数据导出到CSV文件中，比如微软的Excel。

6.2.1 输出结果到CSV

管道和另外一个命令可以在导出文件时派上用场：

```
Get-Process | Export-CSV procs.csv
```

类似于用管道把“Dir”连接到“More”，我们已经把进程信息传输到“Export-CSV”中。第二个Cmdlet有一个强制的位置参数，用于指定输出文件名。因为“Export-CSV”是一个内置的PowerShell Cmdlet，它知道如何把通过“Get-Process”产生的常规表格转换到一个普通的CSV文件中。

现在用Windows记事本打开文件，如图6.2所示。

```
Notepad procs.csv
```

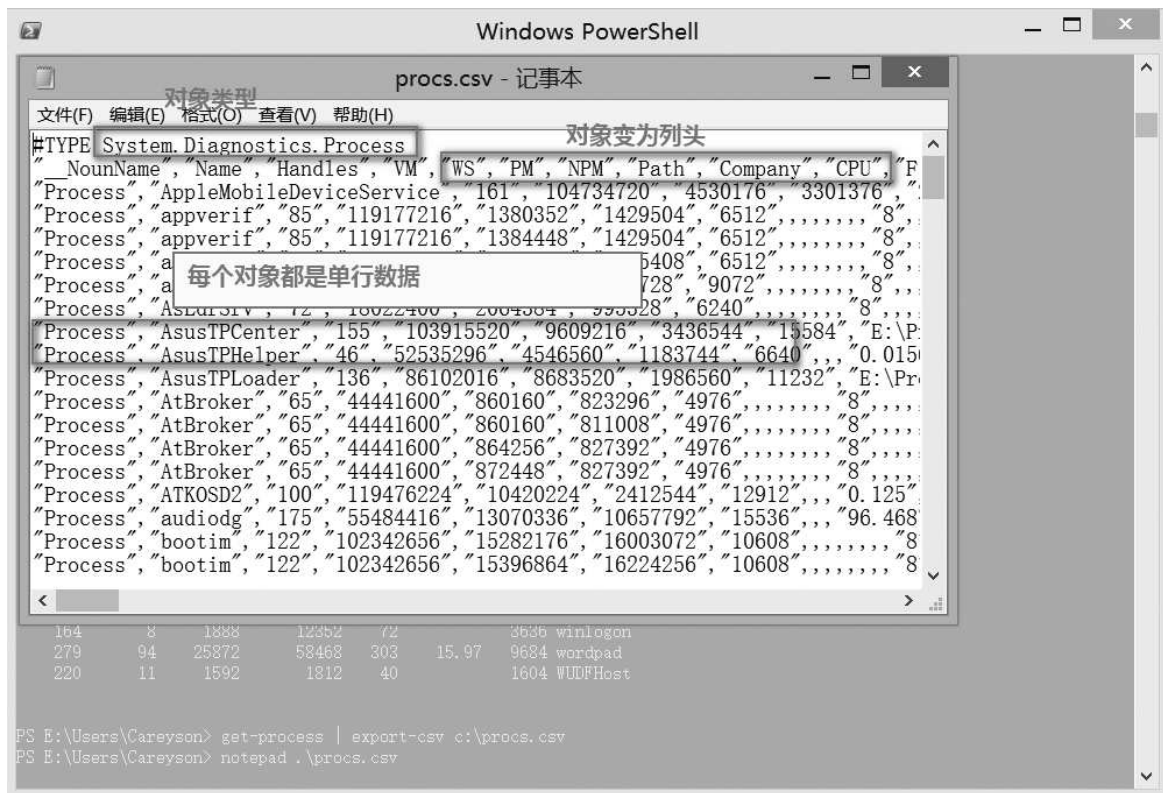


图6.2 在Windows记事本中查看已导出的CSV文件

文件的第一行是以“#”开头的内容，代表着文件中包含的信息类型。以图6.2为例，“System.Diagnostics.Process”是Windows用于标识一个正在运行的进程相关的底层名字。文件的第二行是列名，接下来的每一行代表着每个正在计算机上运行的进程的信息。

你可以把几乎所有的“Get-Cmdlet”用管道传输到“Export-CSV”，然后输出结果。同时，你应该意识到CSV文件包含了比显示到屏幕时更多的信息，因为Shell知道不可能把所有信息全部显示到屏幕中，所以它使用微软提供的配置文件，把最重要的部分显示到屏幕上。在本章的后面，我们会展示如何覆盖配置从而显示你期望的样子。

一旦信息保存到CSV文件，可以轻易地以附件形式发送给同事并让其在PowerShell中查看。只需要用下面的命令把文件导入即可：

```
Import-CSV procs.csv
```

Shell会读取CSV文件然后展示，但是展示的结果并不是和原来格式一样，而是CSV创建时的快照。

6.2.2 输出结果到XML

如果CSV文件不是你想要的，怎么办？没关系，PowerShell还提供了“Export-CliXML”Cmdlet，用于创建常规的命令行界面可扩展标记语言文件（generic command-line interface (CLI) Extensible Markup Language(XML)）。CliXML是PowerShell专用的，但是目前几乎所有程序都能兼容XML。对应的还有“Import-CliXML”Cmdlet。所有的import和export的Cmdlets（比如“Import-CSV”和“Export-CSV”）都强制需要提供文件名作为参数。

动手实验： 尝试导出一些信息如服务、进程或者事件日志到CliXML文件中。确保导出的文件可以用于导入，并且尝试使用记事本和IE浏览器查看这些信息。

除此之外，PowerShell还提供了其他导入导出命令吗？有，可以用“Get-Command”Cmdlet配合“-verb”参数来找到所有“Import”或“Export”的命令。

动手实验： 尝试找一下PowerShell是否自带其他导入导出的Cmdlets。你可以在加载新命令到Shell之后反复尝试，详情请见下一章。

6.2.3 对比文件

在展示、共享信息给别人及后续重新查看过程中，CSV和CliXML文件都很有用。实际上，“Compare-Object”可以在此过程中发挥重要作用。我们会用到它的别名：Diff。

首先，运行“help diff”并阅读相关帮助信息。注意三个参数：-ReferenceObject，-DifferenceObject和-Property。

Diff用于把两个结果集组合一起并进行对比。比如，你在两台不同的机器上运行“Get-Process”。可以把期望用于做匹配的计算机的配置信息放到左边（称为参照计算机）。右边的计算机信息应该尽可能相似（称为差异计算机）。在两边运行命令之后，就可以开始对比两者的信息了，你只需要从中找出它们的差异。

因为这些进程都是类似的，比如你只需要检查类似CPU、内存使用率的值的差异，因此可以忽略一些列。如把注意力放到“Name”列，用于查看

是否包含了多于或少于参照计算机的处理器。如果使用Diff，可以减少你的人工匹配花销。

下面在参照计算机上运行：

```
Get-Process | Export-CliXML reference.xml
```

在这里，我们选择CliXML而不用CSV，是因为CliXML包含了比CSV更多的信息。然后把XML文件传输到差异计算机，运行：

```
Diff -reference (Import-CliXML reference.xml)
➡ -difference (Get-Process) -property Name
```

下面解释一下前面这个比较棘手的步骤：

- 在数学层面上，括号在PowerShell中用于控制执行的顺序。在前面的例子中，强制“Import-CliXML”和“Get-Process”先于“Diff”运行。接着从“Import-CLI”得到的结果被送到“-reference”参数中，而“Get-Process”的结果被送到“-difference”参数中。参数名实际上是“-referenceObject”和“-differenceObject”，在这里你可以提供足够Shell用于识别参数的缩写名即可。也就是本例中的“-reference”和“-difference”已经足够唯一标识这两个参数了。即使我们把这两个参数缩短到“-ref”和“-diff”，命令依旧能运行。
- 相对于匹配两个完整的表格，Diff更加关注“Name”列，所以例子中使用了“-property”这个参数。如果我们不这样定义，结果将全部有差异，因为如“VM”“CPU”和“PM”这些列的值都不一样，结果集是被认为有差异的。
- 匹配结果将以表格形式展示，对于存在于参照结果集但是不存在于差异结果集的数据，会用“<=”标识符表示。对于存在于差异结果集但是不存在于参照结果集的数据，会用“>=”标识符表示。而两者均存在的，则不会出现在“Diff”输出的结果中。

动手实验： 请动手尝试一下，如果手上没有两台电脑，可以把当前信息导出到一个CliXML文件中。然后开启一个新程序，比如记事本、Windows游戏等。再导出数据作为差异结果集，就可以看到效果了。

这是本机的测试结果：

```
PS C:\> diff -reference (import-clixml reference.xml) -difference (get
-process) -property name
```

name	SideIndicator
----	-----
calc	=>
mspaint	=>
notepad	=>
conhost	<=
powerShell_ise	<=

这是一个不错的运维方法，特别是已经建立配置基线，可以对比现有计算机然后找出它们的差异。通过学习这本书，你可以发现很多Cmdlets都能用于运维方面，并且都可以通过管道导出到CliXML文件中以便建立基线。这些基线一般包括服务、进程、操作系统配置、用户及群组等，并且可用于任何时候对比现有系统的差异。

动手实验： 作为尝试，再次执行“Diff”命令，但是不要使用“-property”参数。然后看结果，你会看到每个单独的进程都被列出来，因为如诸如PM/VM等的值都被更改，即使它们是相同的进程。这些输出看上去用处不大，因为它们只显示进程类型名和进程名而已。

顺便一提，“Diff”命令在对比文本文件时并不表现得很好。虽然有些操作系统或者Shell有专门用于匹配文本文件的“Diff”命令，但是PowerShell的“Diff”命令却不一样。你可以在本章的总结实验中体会得到。

注意： 我们希望你多用“Get-Process”“Get-Service”和“Get-EventLog”。这些命令是PowerShell内置的，并且不像Exchange或者SharePoint需要额外的插件才能使用。也就是说，这些技能可以用于以前你学过的所有Cmdlet中，包括Exchange、SharePoint、SQL Server和其他服务器产品。第26章将详细介绍它们。但是目前，请把注意力集中在“如何”使用这些Cmdlets上，而不要过多关注它们的工作原理。我们会在适当的时候加以解释。

6.3 管道传输到文件或打印机

每当你通过“Get-Service”或者“Get-Process”创建一些美观的输出时，你可能想把它们保存到一个文件中甚至纸上。通常来说，Cmdlet是直接输出到PowerShell所在的本地机器的屏幕上，但是你可以修改输出位置。实际上，我们前面已经演示了其中一种方式：

```
Dir > DirectoryList.txt
```

其中“>”符是PowerShell向后兼容旧版本cmd.exe命令的一个快捷方式。而实际上，当你运行这个命令时，PowerShell底层会以下面的方式实现：

```
Dir | Out-File DirectoryList.txt
```

可以自己尝试运行类似的命令，用这种方式替代“>”符号。“Out-File”提供了一些参数让你定制替代的字符编码（如UTF8或Unicode）、追加内容到现有文件等功能。默认情况下，用“Out-File”创建的文件有80列，意味着有时候使用PowerShell需要修改命令的输出，以便适应这80列的限制。这种修改可能导致存到文件的内容格式与使用同样命令显示到屏幕上的不一致。仔细阅读“Out-File”的帮助文档，看看你是否能找到把默认值修改成大于80列的参数。

动手实验：先别看下面的内容，请打开帮助文档看看能否找到答案。我保证你能很快找到。

PowerShell有很多“Out-Cmdlets”，其中一个叫“Out-Default”。它是其中一个不需要额外指定的“Out-Cmdlets”，为什么？请看下面：

当你运行“Dir”时，实际上是在运行“Dir | Out-Default”。“Out-Default”只是把内容指向“Out-Host”，意味着你在无意中运行了：

```
Dir | Out-Default | Out-Host
```

而“Out-Host”是显示结果到显示器中。除此之外，你还找到其他什么“Out-Cmdlets”了吗？

动手实验：是时候研究其他“Out-Cmdlets”了。我们从使用“Help”命令开始，使用如“Help Out*”这样的通配符来获取帮助。这种方式也可以用于“Get-Command”命令，如“Get-Command Out*”，或者指定“-verb”参数：“Get-Command-verb Out”。

“Out-Printer”可能是现有“Out-Cmdlets”中最有用的命令了。虽然“Out-GridView”也有类似功能，但是需要安装.NET Framework v3.5和Windows PowerShell ISE之后才能使用，而这些配置在服务器操作系统中是非自带的。如果你安装了这些，可以尝试运行“Get-Service | Out-GridView”看看结果。“Out-Null”和“Out-String”也非常有用，但是暂时我们不深入探讨。如果你愿意，可以先看看它们的帮助文档。

6.4 转换成HTML

用PowerShell生成HTML报告可行吗？可行。只需要通过管道将结果传递给“ConvertTo-HTML”命令即可。这个命令可以生成结构良好的、通用的HTML数据，并可以在任何Web浏览器中打开。但是这只是原始数据，如果需要美观，需要引用CSS（Cascading Style Sheet）定制样式。注意，这个命令不需要文件名：

```
Get-Service | ConvertTo-HTML
```

动手实验： 确保在阅读本书的时候自己亲手运行命令，我们希望你在理解它们之前先知道它们的功能。

在PowerShell世界里面，动词“Export”意味着你把数据提取，然后转换成其他格式，最后把转换后的格式存到某些存储介质中，如文件。而动词“ConvertTo”仅仅是处理过程的一部分，它仅转换不保存。当你执行前面的命令时，可以看到全屏的HTML数据，明显不是你想要的。那么请思考一下：你应该怎么把HTML存入磁盘的文本文件上？

动手实验： 如果你想到其他方式，尽管尝试。下面的命令就是其中一种：

```
Get-Service | ConvertTo-HTML | Out-File services.html
```

你现在是否看到越来越多强大的命令了？每个命令单独执行一个处理操作，而整个命令行可以被视为一个整体完成一个任务。

PowerShell附带其他“ConvertTo-”Cmdlets，包括“ConvertTo-CSV”和“ConvertTo-XML”等。正如“ConvertTo-HTML”一样，这些命令都不在磁盘上创建文件，只是把命令的输出分别转换成CSV或XML。你需要用管道把它们和“Out-File”连接起来以便存储到磁盘上，但是它们比使用“Export-CSV”或“Export-CliXML”更简短。另外，它们能既转换又存储。

补充说明

现在闲聊一些背景知识。在本例中，经常有学生问：为什么微软提供了“Export-CSV”和“ConvertTo-CSV”这两个对于XML数据来说看上

去几乎一样的功能？

在某些高级场景中，你可能不想把结果存到磁盘文件上。比如你想把数据转换成XML然后传输到Web服务，或者其他地方。通过使用不需要存储文件的“ConvertTo-”Cmdlets，你可以灵活地实现你的需求。

6.5 使用Cmdlets修改系统：终止进程和停止服务

导出和转换不是你希望连接两个命令的唯一目的。比如下面的例子，记住不要运行：

```
Get-Process | Stop-Process
```

你能想象一下使用这个命令会怎样吗？会宕机！它会检索每一个进程，然后尝试逐个终止。这是一个很危险的进程，类似本地安全权限（Local Security Authority），你的电脑很可能进入蓝屏死机状态。如果你在虚拟机中运行PowerShell倒是可以尝试一下。

这个例子想说明的是带有相同名词（本例中的进程）的Cmdlets可以在彼此之间互传信息。通常情况下，你最好带上特定进程名而不是终止全部：

```
Get-Process -name Notepad | Stop-Process
```

服务也是类似的，“Get-Service”命令的输出结果能和其他Cmdlets（如Stop-Service、Start-Service、Set-Service等）一起被管道传输。

你可能想象得到，命令之间能互相连接是需要符合某些特定规则的。比如，当你看到这样：Get-ADUser | New-SQLDatabase的指令序列，你会知道它不会实现什么有意义的功能（虽然它的确做了一些无用功）。在第7章中，我们会深入解释这些管理命令间互相连接的规则。

下面我们希望你对类似“Stop-Service”和“Stop-Process”这些Cmdlets有更深入的了解。这些Cmdlets以某些方式修改系统，并且有一个内部定义的影响级别（impact level）。Cmdlet的创建者已经设定了这些影响级别，并且不允许修改。而Shell有一个相应的“\$ConfirmPreference”设置，默认为“High”。可以通过下面的命令查看你的Shell的设置：

```
PS C:\> $ConfirmPreference
High
```

工作原理：当Cmdlet的内部影响级别大于等于Shell的“\$ConfirmPreference”设置时，不管Cmdlet正准备做什么，Shell都会自动询问“你确定要这样做吗？（Are you sure？）”。实际上，如果你使用虚拟机尝试前面提到的那个“宕机”命令，你会发现对于每个进程，都会问一次“Are you sure?”。当Cmdlet的内部影响级别小于Shell的“\$ConfirmPreference”设置时，不会自动弹出这个提示。

但是如果你“喜欢”它总是弹出，可以使用下面的命令：

```
Get-Service | Stop-Service -confirm
```

我们在这里加了“-confirm”参数，对于某些被支持的用于修改系统的Cmdlet，会弹出提示，并对这些被支持的Cmdlet显示对应的帮助文档。

另外一个类似的参数是“-whatif”，可用于支持“-confirm”的Cmdlet。但是它并不默认触发，可以在你想用的时候使用：

```
PS C:\> get-process | stop-process -whatif
What if: Performing operation "Stop-Process" on Target "conhost (1920)"
".
What if: Performing operation "Stop-Process" on Target "conhost (1960)"
".
What if: Performing operation "Stop-Process" on Target "conhost (2460)"
".
What if: Performing operation "Stop-Process" on Target "csrss (316)".
```

它会告诉你哪些Cmdlet会被执行，但是并不真正运行。这个功能为那些可能有潜在风险的Cmdlet的预览提供了很好的帮助，并且可以检查是否是你想要的结果。

6.6 常见误区

在PowerShell中，其中一个常见的困惑是“Export-CSV”和“Export-CliXML”的异同。这两个命令从技术上都是用于创建文本文件。也就是

说，两者的输出结果都能在记事本中查看，如图6.2所示。但是你必须承认两个结果有明显的差异——一个是逗号分隔值，而另外一个则是XML。

这个问题主要关心的是用户如何把文件重复读入Shell中。为此你是否使用“Get-Content”（或者它的别名，Type/Cat）？举个例子，假设你这样使用：

```
PS C:\> get-eventlog -LogName security -newest 5 | export-csv
events.csv
```

现在你需要使用“Get-Content”命令从Shell中读出来：

```
PS C:\> Get-Content .\events.csv
#TYPE System.Diagnostics.EventLogEntry#security/Microsoft-Windows-
Security
-Auditing/4797
"EventID", "MachineName", "Data", "Index", "Category", "CategoryNumber",
"EntryT
ype", "Message", "Source", "ReplacementStrings", "InstanceId", "TimeGener
ated",
"TimeWritten", "UserName", "Site", "Container"
"4797", "DONJONES1D96", "System.Byte[]", "263", "
(13824)", "13824", "SuccessAudi
t", "An attempt was made to query the existence of a blank password for
an
account.

Subject:
      Security ID:          S-1-5-21-87969579-3210054174-450162487-
100

      Account Name:         donjones
      Account Domain:       DONJONES1D96
      Logon ID:              0x10526

Additional Information:
      Caller Workstation:    DONJONES1D96
      Target Account Name:   Guest
      Target Account Domain: DONJONES1D96", "Microsoft-Windows-
Security-
uditing
", "System.String[]", "4797", "3/29/2012 9:43:36 AM", "3/29/2012 9:43:36
AM", ,
,
"4616", "DONJONES1D96", "System.Byte[]", "262", "
(12288)", "12288", "SuccessAudi
```

```
t", "The system time was changed.
```

我们截断了前面的输出，但是还是可以看到有很多相同的部分。回顾原始的CSV数据，你是否觉得有很多垃圾信息？该命令没有尝试解析、编译这些数据。现在对比一下“Import-CSV”的结果：

```
PS C:\> import-csv .\events.csv

EventID           : 4797
MachineName       : DONJONES1D96
Data              : System.Byte[]
Index             : 263
Category          : (13824)
CategoryNumber    : 13824
EntryType         : SuccessAudit
Message           : An attempt was made to query the existence of a
                   blank password for an account.

                   Subject:
                       Security ID:
                       S-1-5-21-87969579-3210054174-450162487-1001
                       Account Name:      donjones
                       Account Domain:    DONJONES1D96
                       Logon ID:          0x10526
                   Additional Information:
                       Caller Workstation: DONJONES1D96
                       Target Account Name: Guest
                       Target Account Domain: DONJONES1D96
Source            : Microsoft-Windows-Security-Auditing
ReplacementStrings : System.String[]
InstanceId        : 4797
TimeGenerated     : 3/29/2012 9:43:36 AM
TimeWritten       : 3/29/2012 9:43:36 AM
UserName          :
```

是不是好很多？“Import-Cmdlets”会关注文件中的内容，尝试解析它们，然后创建一个比原始命令（本例中的“Get-EventLog”）看上去更加顺眼的输出结果。如果你使用“Export-CSV”创建文件，可以使用“Import-CSV”命令来读取它们。如果使用“Export-CliXML”命令创建文件，通常建议使用“Import-CliXML”命令读取。使用这些配套命令可以得到更好的结果。仅在从一个文本文件中读取内容并且不需要PowerShell解析数据时，才使用“Get-Content”命令，也就是你仅需要原始内容。

6.7 动手实验

注意:

本实验需要PowerShell v3或以上版本。

由于前面演示的例子稍微花时间，所以我们尽可能保证本章文字的简洁，因为我们希望你能把更多精力花在下面的动手实验中。如果你还没完成本章中所有“动手实验”的任务，我们强烈建议你先去完成它们，然后进行下面的任务：

1. 在控制台运行“`Get-Service | Export-CSV services.csv | Out-File`”时会发生什么情况？为什么会这样？

2. 除了获取一个或多个服务及以管道方式传输到“`Stop-Service`”之外，“`Stop-Service`”服务还提供了其他什么方式让你指定服务或停止服务？有什么方式可以在不使用“`Get-Service`”的前提下停止一个服务？

3. 如何创建一个竖线分隔符文件替代一个逗号分隔符（CSV）文件？你可以依旧使用“`Export-CSV`”命令，但是应该使用什么参数？

4. 可以在已导出的CSV文件头部忽略#命令行吗？这一行通常包含了类型信息，但是如果你想从一个特定文件中获取并忽略时要怎么做？

5. “`Export-CliXML`”和“`Export-CSV`”都可以通过创建并覆盖文件来修改系统，你可以用什么参数来阻止它们覆盖现有文件？还有什么参数可以在你输出文件前提醒并请求确认？

6. Windows维护少数局部配置，包括一个默认分隔符列表。在美国系统中，分隔符是逗号。你如何让“`Export-CSV`”使用当前系统默认的分隔符而不是逗号？

动手实验： 完成上面实验之后，尝试完成本书附录中的实验回顾

1。

第7章 扩展命令

可扩展性是PowerShell的一个主要优势。随着微软对PowerShell的持续投入，它为Exchange Server、SharePoint Server、System Center系列、SQL Server等产品开发了越来越多的命令。通常，当你安装这些产品的管理工具时，还会安装一个或多个Windows PowerShell扩展的图形化管理控制台。

7.1 如何让一个Shell完成所有事情

我们知道你可能熟悉图形化的微软管理控制台（MMC），这就是为什么我们将使用它作为例子讲述PowerShell是如何工作的。它们涉及的可扩展的工作原理是一样的，部分原因是MMC和PowerShell由同一个管理框架下的团队所研发。

当你打开一个新的空白MMC控制台，在很大程度上，它的功能是有限的。因为MMC的内置功能很少，所以它基本上做不了什么事情。如果想让它强大一些，你需要在文件菜单中使用添加/删除管理单元。在MMC中，一个管理单元就是一个工具，这类似于活动目录用户和计算机、DNS管理、DHCP管理等。你可以在MMC中添加你喜欢的管理单元，也可以保存生成控制台，这使得下次更加方便地重新打开同一套管理单元。

这些管理单元从何而来？一旦你安装了类似Exchange Server、Forefront或者System Center产品的相关管理工具，会在MMC的添加、删除管理单元的对话框里面列出这些产品的管理单元。大多数产品也安装自己的预配置MMC控制台文件，它什么也不做，只是加载了基本的MMC和预加载一个或两个管理单元。如果你不想，可以不必使用这些预配置控制台，因为你总能打开一个空白的MMC控制台，并加载你需要的管理单元。例如，预配置的Exchange Server MMC控制台不包括活动目录站点和服务的管理单元，但你可以很容易地创建一个MMC控制台，包括交易所，也是站点和服务。

PowerShell的工作原理方式几乎与MMC完全一样。安装一个给定产品的管理工具（安装管理工具的选项通常包含在产品的安装菜单中。如果你在Windows 7上安装类似Exchange Server的产品，它的安装只提供了该管理工具）。这样做会为你提供PowerShell的相关扩展，它甚至可能会创建该产品特定的Shell管理程序。

7.2 关于产品的“管理Shell”

这些管理特定产品的Shell程序的来源很混乱。我们必须澄清：只有一个Windows PowerShell。根本就没有分Exchange PowerShell和活动目录PowerShell，只有一个Shell。

以活动目录为例，在Windows Server 2008 R2域控制器的开始菜单、管理工具下，你会发现一个关于活动目录组件的Windows PowerShell。如果在这一项单击右键，然后从上下文菜单中选择属性，第一眼就可以看到类似如下的目标域：

```
%windir%\system32\WindowsPowerShell\v1.0\powershell.exe  
➔-noexit -command import-module ActiveDirectory
```

该命令运行标准的PowerShell.exe应用程序，而且指定命令行参数运行特定命令：**Import-Module ActiveDirectory**。执行的效果是可以预加载活动目录。但是，我们没有理由会认为：为什么不能打开“正常”的PowerShell并运行相同的命令获得相同的功能。

你可以找到同样适用于几乎所有特定于产品的“管理Shell”：**Exchange**、**SharePoint**等。查看这些产品开始菜单快捷方式的属性，你会发现，它们都是打开标准的PowerShell.exe，并以传递一个命令行参数的方式来添加一个模块、增加一个管理单元或者加载一个预配置控制台文件（该控制台文件是一个包含需要自动加载管理单元的简单列表）。

SQL Server 2008 和 **SQL Server 2008 R2**却是例外。它们“产品特定”Shell叫作**Sqlps**。它是一个经过特殊编译专门运行SQL Server扩展的PowerShell。通常称之为**mini-Shell**。微软第一次在SQL Server尝试这种方法。但这种方法已经不流行了，并且微软不会再使用这种方法了：**SQL Server 2012**使用的是**PowerShell**。

你不只局限于使用预先设定的扩展。当你打开**Exchange**的管理Shell程序，你可以运行**Import-Module ActiveDirectory**并假设该活动目录模块已经存在于你的电脑，添加活动目录功能到Shell中。你也可以打开标准的PowerShell控制台并手动添加你想要的扩展。

正如这一节前面提到的，这是一个让人感到非常困惑的知识点，包括有些人认为多个版本的PowerShell不能交叉利用彼此的功能。**Don**（作者）甚至在他的博客（<http://windowsitpro.com/go/DonJonesPowerShell>）中进行了讨论。**PowerShell**团队成员介入并支持他，所以，请相信我们：你可以在

一个Shell中包含所有你想要的功能，而在开始菜单中，特定产品的快捷方式不会以任何方式限制或暗示你这些产品存在特殊版本的PowerShell。

7.3 扩展：找到并添加插件

PowerShell存在两种类型的扩展：模块和管理单元。首先讲述管理单元。

一个适合管理单元PowerShell的名字是PSSnapin，用于区别这些来自管理单元的图形MMS。PSSnapins在PowerShell v1版本的时候就已经存在了。一个PSSnapin通常包含一个或多个DLL文件，同时包含配置设置的XML文件和帮助文档。PSSnapins必须先安装和注册，然后PowerShell才能识别它的存在。

注意： PSSnapin的概念逐步被微软移除了，将来可能会越来越少出现。在内部，微软的重点是提供扩展的模块。

你可以通过在PowerShell中运行Get-PSSnapin -registered命令获取到一个可用的管理单元列表。因为我在域控制机器上安装了SQL Server 2008，所以执行命令返回的结果如下：

```
PS C:\> get-pssnapin -registered

Name       : SqlServerCmdletSnapin100
PSVersion  : 2.0
Description : This is a PowerShell snap-in that includes various SQL
              Server Cmdlets.
Name       : SqlServerProviderSnapin100
PSVersion  : 2.0
Description : SQL Server Provider
```

上面的信息说明我的机器上安装了两个可用的管理单元，但是并没有加载。你可以通过运行Get-PSSnapin命令来查看加载的列表。该列表包含所有的核心，自动加载的管理单元包含PowerShell中的本机功能。

通过运行Add-PSSnapin并指定管理单元名的方式来加载某一个管理单元：

```
PS C:\> add-pssnapin sqlserverCmdletsnapin100
```

类似常用的PowerShell命令，你不必担心大小写是否正确，Shell是忽略大小写的。

当一个管理单元加载成功了，你可能想知道Shell到底增加了什么功能。PSSnapin可以增加Cmdlets命令、提供PSDrive，或者两者都增加。使用Get-Command（或者别名：Gcm）命令找出增加的Cmdlets命令：

```
PS C:\> gcm -pssnapin sqlserverCmdletsnapin100
```

CommandType	Name	Definition
-----	----	-----
Cmdlet	Invoke-PolicyEvaluation	Invoke-PolicyEvaluation...
Cmdlet	Invoke-Sqlcmd	Invoke-Sqlcmd [[-Query]...

我们在这里必须指出，输出的结果中只包含了SqlServerCmdletSnapin100这个管理单元，并且只有两行记录。是的，这就是SQL Server在管理单元中增加的所有内容，而且只有一个可以执行Transact-SQL(T-SQL)的命令。因为你可以通过T-SQL命令在SQL Server上实现几乎所有的操作，Invoke-Sqlcmd这个Cmdlet命令同样可以完成所有的操作。

运行Get-PSProvider可以查看一个管理单元提供哪些新的PSDrive，你不能在该Cmdlet命令指定某个管理单元，所以你必须熟悉哪些提供程序已经存在，并通过查看列表方式发现新增内容。下面是返回结果：

```
PS C:\> get-psprovider
```

Name	Capabilities	Drives
----	-----	-----
WSMan	Credentials	{WSMan}
Alias	ShouldProcess	{Alias}
Environment	ShouldProcess	{Env}
FileSystem	Filter, ShouldProcess	{C, A, D}
Function	ShouldProcess	{Function}
Registry	ShouldProcess, Transa...	{HKLM, HKCU}
Variable	ShouldProcess	{Variable}
Certificate	ShouldProcess	{cert}

看起来没有任何新增内容。我们并不感到惊讶，因为管理单元是通过SqlServer CmdletSnapin100名称来加载的。如果你回忆一下，我们的可用管

理单元同样包含了SqlServerProviderSnapin100，这意味着微软出于某些原因，把它的Cmdlets命令和PSDrive分开打包。让我们尝试添加第二个：

```
PS C:\> add-pssnapin sqlserverprovidersnapin100
PS C:\> get-psprovider
```

Name	Capabilities	Drives
WSMan	Credentials	{WSMan}
Alias	ShouldProcess	{Alias}
Environment	ShouldProcess	{Env}
FileSystem	Filter, ShouldProcess	{C, A, D}
Function	ShouldProcess	{Function}
Registry	ShouldProcess, Transa...	{HKLM, HKCU}
Variable	ShouldProcess	{Variable}
Certificate	ShouldProcess	{cert}
SqlServer	Credentials	{SQLSERVER}

回顾一下之前的输出结果，可以发现SQL Server驱动器已经被添加到我们的Shell当中，由SQL Server的PSDrive提供驱动。新增的该驱动意味着可以运行命令：cd SQL server切换到SQL Server驱动器，接着可以开始探索数据库。

7.4 扩展：找到并添加模块

PowerShell提供的第二种扩展方式叫作模块。模块被设计得更加独立，因此更加容易分发，但是它的工作原理类似于PSSnapins。但是，你需对它们有更多了解，这样才能找到和使用它们。

模块不需要复杂的注册。反而，PowerShell会自动在一个特定的目录下查找模块。PSModulePath这个环境变量定义了PowerShell期望存放模块的路径：

```
PS C:\> get-content env:psmodulepath
C:\Users\Administrator\Documents\WindowsPowerShell\Modules;C:\Windows\system32\WindowsPowerShell\v1.0\Modules\
```

在前面的例子中可以发现，路径中包含了两个默认的位置：其中一个存放系统模块的操作系统目录，另外一个存放个人模块的文档目录。只要你知道一个模块的完整路径，你也可以从任何其他的位置添加模块。

注意： PSModulePath并不能在PowerShell中修改，它是你操作系统环境变量的一部分。你可以在系统控制面板对它进行修改，或者通过组策略。

在PowerShell中，该路径很重要。如果你有位于其他位置的模块，你应该把模块所在的路径加入到PSModulePath这个环境变量中。图7.1展示了如何通过系统控制面板而不是PowerShell去修改该环境变量。

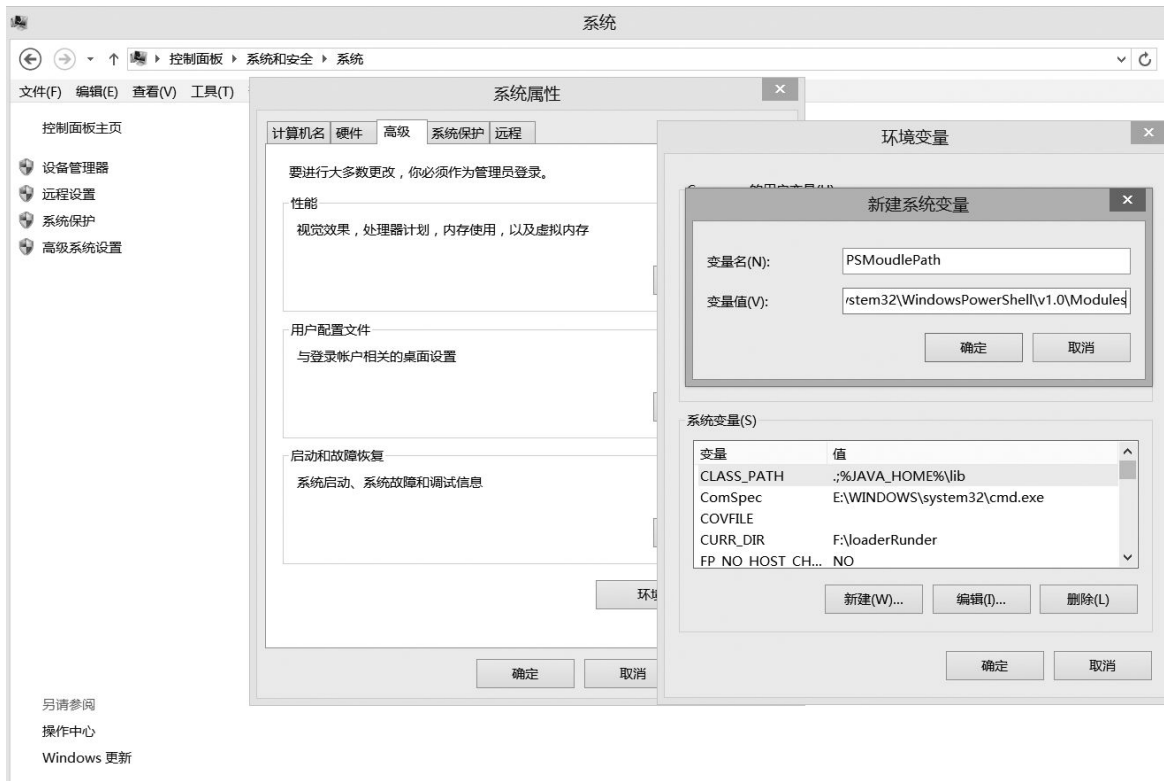


图7.1 修改Windows下的PSModulePath环境变量

为什么PSModulePath这个环境变量的路径如此重要？因为通过它，PowerShell可以自动加载位于你计算机上的所有模块。PowerShell会自动发现这些模块。换句话说，它看起来好像是所有的模块都已被加载了。查看一个模块的帮助，会发现你不需要手动加载它。运行任何的命令，PowerShell都会自动加载该命令相关的模块。PowerShell的Update-Help命令同样使用PSModulePath发现存在的任何模块，然后针对每个模块搜索需要更新的帮助文档。

例如，运行Get-Module | Remove-Module移除所有加载的模块。接着运行下面的命令（你返回的结果可能会有细微的差异，这取决于你所使用的Windows版本）：

```
PS C:\> help *network*
```

Name	Category	Module
Get-BCNetworkConfiguration	Function	BranchCache
Get-DtcNetworkSetting	Function	MsDtc
Set-DtcNetworkSetting	Function	MsDtc
Get-SmbServerNetworkInterface	Function	SmbShare
Get-SmbClientNetworkInterface	Function	SmbShare

正如你所看到的，PowerShell发现了几个命令名中包含“network”关键字的命令（在函数分类里）。即使你没有加载该模块，你也可以查看它们中任何一个的帮助信息：

```
PS C:\> help Get-SmbServerNetworkInterface
```

名称

Get-SmbServerNetworkInterface

语法

```
Get-SmbServerNetworkInterface [-CimSession <CimSession[]>]  
[-ThrottleLimit <int>] [-AsJob] [<CommonParameters>]
```

如果你想，你甚至可以运行该命令，PowerShell确保会自动为你加载该模块。这个自动发现和自动加载的功能非常有用，甚至帮你发现和使用你在启动Shell时没有出现的命令。

提示：

你也可以使用Get-Module命令检索一个远程服务器的可用模块列表，还可以使用Import-Module加载一个远程模块到当前PowerShell会话。你将在第13章中的远程控制中学习到如何使用该功能。

即使在模块还没有显式地加载到内存的情况下，PowerShell依然可以自动发现模块让Shell完成命令名称自动补全（在控制台使用Tab按钮，或者使用ISE的智能提醒）、显示帮助和运行命令。该特性使得保持PSModulePath环境变量的完整和最新很有必要。

如果一个模块不在被PSModulePath引用的任何一个目录下，你应该使用Import-Module命令并指定模块的完整路径，如C:\MyPrograms\Something\MyModule。

如果在开始菜单有一个特定产品Shell的快捷方式，比如说Share Point Server，而你却不知道该产品安装PowerShell模块的路径，打开快捷方式图

标的属性，像本章之前教你的方法，在快捷方式的目标属性中会包含使用 **Import-Module** 命令需要的模块名和路径。

模块还可以添加 **PSDrive**。你必须使用在 **PSSnapins** 中相同的技巧来确定有哪些新的提供者：运行 **Get-PSProvider** 命令。

7.5 命令冲突和移除扩展

仔细看看我们为 **SQL Server** 和活动目录增加的命令。注意到什么特别的命令名了吗？

大多数的 **PowerShell** 扩展（**Exchange Server** 是一个明显的例外）都在它们命令名的名词部分增加了一个短的前缀，如 **Get-ADUser** 和 **Invoke-SqlCmd**。这些前缀看起来有些多余，但是它们可以防止命令名的冲突。

例如，假设你加载的两个模块中都包含了 **Get-User** 这个 **Cmdlet** 命令。这样两个命令名称相同，且被同时加载。你运行 **Get-User** 时，**PowerShell** 应该执行哪个呢？事实上是执行最后一个加载模块的命令。但是另外一个相同的命令却无法被访问。为了明确所需运行的具体命令，你需要使用看起来有点多余的命名规则，它包括管理单元名称和命令名称。如果 **Get-User** 来自一个叫作 **MyCoolPowerShellSnapin** 的模块单元，你需要使用下面的方式运行：

```
MyCoolPowerShellSnapin\Get-User
```

这需要输入很多内容，这就是为什么微软建议添加特定产品前缀，如在每个命令的名词中加入 **AD** 或者 **SQL**。增加前缀可以防止冲突，并且使命令更容易识别和使用。

如果你已经对冲突不厌其烦，你可以随时选择删除冲突的扩展名。你需要运行 **Remove-PSSnapin** 或 **Remove-Module**，并指定管理模块或模块命令的名称，从而卸载某个扩展。

7.6 玩转一个新的模块

让我们开始对刚刚学习到的新知识加以实践。假设你使用最新版本的 **Windows** 系统，并希望你能跟随我们目前在本节的命令。更重要的是，我们希望你能跟随该过程并思考我们将要解释的内容，因为这是我们教自己如何使用遇到的新命令而没有冲出去为每个单独的产品和功能买一本新书的方

法。在本章的后面的动手实验，我们会让你自己重复该过程来学习一个全新的命令集。

我们的目标是清除我们计算机上的DNS名称解析缓存。我们还不知道PowerShell是否能做到这一点，所以我们先要在帮助系统中寻找一些线索：

```
PS C:\> help *dns*

Name                Category      Module
-----
dnsn                Alias
Resolve-DnsName     Cmdlet        DnsClient
Clear-DnsClientCache Function       DnsClient
Get-DnsClient        Function       DnsClient
Get-DnsClientCache   Function       DnsClient
Get-DnsClientGlobalSetting Function       DnsClient
Get-DnsClientServerAddress Function       DnsClient
Register-DnsClient   Function       DnsClient
Set-DnsClient         Function       DnsClient
Set-DnsClientGlobalSetting Function       DnsClient
Set-DnsClientServerAddress Function       DnsClient
Add-DnsClientNrptRule Function       DnsClient
Get-DnsClientNrptPolicy Function       DnsClient
Get-DnsClientNrptGlobal Function       DnsClient
Get-DnsClientNrptRule Function       DnsClient
Remove-DnsClientNrptRule Function       DnsClient
Set-DnsClientNrptGlobal Function       DnsClient
Set-DnsClientNrptRule Function       DnsClient
```

是的！正如你看到的，这就是我们计算机上所有的DnsClient模块。前面的列表中显示了Clear-DnsClientCache命令，但是我们好奇哪个命令可用。为了找出该命令，我们手动加载该模块并列出所有命令。

动手实验： 继续跟随我们运行这些命令。如果你计算机上没有DnsClient这个模块，那是因为你使用了一个较旧的Windows版本。请考虑获取一个新的版本，甚至是在你的虚拟机里运行一个实验版本，直到可以运行下述命令：

```
PS C:\> import-module -Name DnsClient
PS C:\> get-command -Module DnsClient

Capability  Name
-----
CIM         Add-DnsClientNrptRule
CIM         Clear-DnsClientCache
```

```
CIM      Get-DnsClient
CIM      Get-DnsClientCache
CIM      Get-DnsClientGlobalSetting
CIM      Get-DnsClientNrptGlobal
CIM      Get-DnsClientNrptPolicy
CIM      Get-DnsClientNrptRule
CIM      Get-DnsClientServerAddress
CIM      Register-DnsClient
CIM      Remove-DnsClientNrptRule
CIM      Set-DnsClient
CIM      Set-DnsClientGlobalSetting
CIM      Set-DnsClientNrptGlobal
CIM      Set-DnsClientNrptRule
CIM      Set-DnsClientServerAddress
Cmdlet   Resolve-DnsName
```

注意： 可以查看关于Clear-DnsClientCache的帮助，或者甚至直接运行命令。PowerShell会在后台为我们加载DnsClient模块。因为我们正处于探索阶段，这种方法可以查看到该模块的完整命令列表。

该命令列表看起来跟我们之前的列表或多或少有些相似。好的，让我们来看看Clear-DnsClientCache命令：

```
PS C:\> help Clear-DnsClientCache

名称
    Clear-DnsClientCache
语法
    Clear-DnsClientCache [-CimSession <CimSession[]>] [-ThrottleLimit
    <int>] [-AsJob] [-WhatIf] [-Confirm] [<CommonParameters>]
```

看起来已经很明确了，我们没有发现任何强制参数。让我们尝试运行命令：

```
PS C:\> Clear-DnsClientCache
```

好的，通常来说，没有消息就是最好的消息。尽管如此，我们更愿意看到该命令到底做了什么事情。尝试使用下面的命令：

```
PS C:\> Clear-DnsClientCache -verbose
详细信息: The specified name resolution records cached on this machine
will be removed.
```

```
Subsequent name resolutions may return up-to-date information.
```

这个`-verbose`开关虽然不会输出所有命令，但是对所有的命令都有效。在该示例中，我们得到一个指示发生了什么事情的信息，这让我们知道这个命令已经成功运行了。

7.7 配置脚本：在启动Shell时预加载扩展

假设你已经打开了PowerShell，并且你已经加载了几个你最为喜欢的管理单元和模块。如果你接受这种方式，你需要为每个管理单元或模块运行一个个的命令。如果你有几个需要装载的话，这会花费几分钟来输入命令。当你不想使用Shell而关闭了它的窗口时，下次重新打开Shell窗口，之前加载的管理单元和模块都不复存在了，而你需要运行命令重新加载它们。这是件多么可怕的事情。肯定有一个更好的方式可以解决该问题。

我们给你介绍3种更好的方式。第一个涉及创建一个控制台文件。这只能记录已经加载的PSSnapins，对已经加载的模块是不起作用的。首先加载所有你想要的管理单元，接着运行下面的命令：

```
Export-Console c:\myShell.psc
```

运行该命令，可以把你在Shell中加载的管理单元列表保存到一个很小的XML文件。

接下来，你希望在某些地方创建一个新的PowerShell快捷方式，快捷方式的目标应该是：

```
%windir%\system32\WindowsPowerShell\v1.0\powershell.exe  
➡ -noexit -psconsolefile c:\myShell.psc
```

当你使用该快捷方式打开一个新的PowerShell窗口，这将加载控制台，并且该Shell会自动加载控制台文件里面列表中的所有管理单元。再次提醒，不能包括模块。如果同时存在管理单元和模块或者你只想加载其中某些模块，这种情况下你应该怎么做呢？

提示： 请记住，PowerShell会自动加载PSModulePath环境变量的其中一个路径中的模块。如果你想预加载模块，你只需要考虑该模块是否存在PSModulePath环境变量的其中一个路径

中。

答案就是使用配置脚本。我们在前面提到，将在本书的第25章进行详细的讨论。现在按照下面的步骤来学习如何使用它们：

1. 在你的文档目录创建一个名为**WindowsPowerShell**（在文件夹名中不要包含空格）的新文件夹。

2. 在上面创建的文件夹中使用记事本创建一个名为**profile.ps1**的新文件。当你使用记事本保存该文件时，需要确保文件名使用引号括起来（“**profile.ps1**”）。使用引号是为了防止记事本在文件名加上.txt的文件扩展名。如果加上了.txt扩展名，这种方法就行不通了。

3. 在刚刚创建的文本文件输入**Add-PSSnapin**和**Import-Module**命令，以一行一个命令的格式来加载管理单元和模块。

4. 回到**PowerShell**中，你需要启用脚本的执行功能，这在默认情况下是禁用的。我们将会在第17章讨论这样操作带来的安全隐患，但是现在我们假设你是在一个单独的虚拟机或者是单独的测试机上做该操作的，这样安全性就不再是个问题了。在该脚本中，运行**Set-ExecutionPolicy RemoteSigned**命令。需要注意的是，该命令只有在你以管理员身份运行**Shell**的时候才会执行。也可以使用组策略对象（**GPO**）来覆盖该设置。如果是这样做，你会得到一个警告消息。

5. 假设到目前为止你没有收到任何的错误或者警告。关闭并重启**Shell**，这将会自动加载**profile.ps1**文件，执行里面的命令，为你加载喜欢的管理单元和模块。

动手实验： 如果你没有找到一个喜欢的管理单元或模块，创建上面这个简单的配置文件将是一个很好的练习。如果实在不知道输入什么，可以在配置脚本输入**cd **，这样你每次打开**Shell**的时候就会跳转到系统盘的根目录。但不要在你生产环境中的机器上执行上面的操作，因为我们还没有解决所有的安全隐患。

7.8 常见误区

使用**PowerShell**的新手，当他们开始操作模块和管理单元时经常会做一件错误的事情：他们不阅读帮助文档。特别地，他们在查看帮助的时候不使用**-example**或者**-full**开关。

坦白说，查看内建的示例是学习使用一个命令最好的方式。是的，滚动数以百计的命令列表可能是有点吓人（如Exchange Server，新增的命令大大超过了400个），但是通过在命令Help和 Get-Command基础上加通配符应该可以更容易缩小列表的范围。因此，阅读帮助文档吧！

7.9 动手实验

注意： 在本实验中，你需要一个Windows 7、Windows Server 2008 R2或者是更高版本的操作系统来运行PowerShell v3甚至是更高的版本。

通常，我们假设在你的计算机或者虚拟机上的操作系统为最新版本（客户端或者服务器版本）来运行测试。

在本实验，你只有一个任务：运行网络故障诊断包。当你成功做到了，你需要寻找“实例ID”敲入回车键，运行Web连接测试，并且从一个指定的页面中寻求帮助。使用<http://videotraining.interfacett.com> 作为你的测试地址。我们希望你获取的返回信息是“没有发现问题”，这意味着你运行该检查成功了。

为了完成该任务，你需要找到一个可以获取到故障诊断包的命令，并且需要一个可以执行故障诊断包的命令。你还需要找到这些包所处的位置和它们的名字。你需要知道的所有内容都在PowerShell里，帮助系统将为你找到它们。

这是你得到的所有帮助！

第8章 对象：数据的另一个名称

在本章我们将会尝试做一些不同的事情。我们发现PowerShell中对于对象的使用是最让人困惑的内容之一，但同时也是Shell中最关键的内容，影响在Shell中的所有操作。这些年我们尝试通过不同的方式对该概念进行阐述，最终我们找到了能够让完全不同背景的受众都能接受的阐述方式。如果你之前曾有过编程经验并因此很容易能够接受对象的概念，请跳过8.2节。如果你没有编程背景且没有在脚本语言或编程语言中使用过对象，请从8.1节开始阅读本章。

8.1 什么是对象

花一点时间运行PowerShell中的Get-Process。可以看到一个包含多列的表格，但这些信息仅仅是关于进程的冰山一角。进程对象还包括机器名、主窗口句柄、最大工作集大小、退出代码和时间、处理器掩码信息以及其他大量信息。实际上，你可以找出超过60个与进程有关的信息。为什么PowerShell仅仅展示少量的信息呢？

原因非常简单，PowerShell当然可以提供屏幕上所无法容纳的更多的信息。当运行任意命令，比如Get-Process、Get-Service、Get-EventLog或其他命令时，PowerShell会完全在内存中构造用于容纳关于项的所有信息的表格。例如Get-Process，该表格由67列组成，每行对应运行在计算机中的一个进程。每一列包含一部分信息，比如说虚拟内存、CPU利用率、处理器名称、进程ID等。然后，PowerShell会检查你是否指定所需查看的列。如果你未指定（目前我们还没展示如何指定）想查看的列，Shell会查看由微软提供的配置文件并只显示微软认为你希望查看的列。

一种查看所有列的方式是使用ConvertTo-HTML命令：

```
Get-Process | ConvertTo-HTML | Out-File processes.html
```

该命令不会过滤列，而是生成包含所有列的HTML文件。这是查看整个表的一种方式。

除去包含这些信息的列之外，表中每一行都有一些与之对应的方法。这些方法包括操作系统可以对进程进行的操作。比如说，操作系统可以关闭进程、杀死进程、刷新信息，或者等待进程退出等。

每当运行一个可以产生结果的命令时，输出结果在内存中以表的形式存放。当将输出结果以管道的方式由一个命令传送给另一个命令时，比如说：

```
Get-Process | ConvertTo-HTML
```

整个表通过管道进行传输。该表在传输过程中并不会过滤到只有一小部分列，而是直到所有的命令都运行后才会进行过滤。

下面是一些术语的变化。**PowerShell**并不会将这些内存中的表命名为“表”，而是使用下述4个术语：

- 对象——这也就是所谓的“表行”。它代表单个事物，比如说单个进程或是单个服务。
- 属性——这也就是所谓的“表列”。它代表关于对象的一部分信息，比如说进程名称、进程ID或服务状态。
- 方法——这也就是所谓的“行为”。方法与某个对象关联并使得对象完成某些任务，比如说杀死进程或启动服务。
- 集合——这是整个对象的集合，我们曾称之为“表”。

如果你发现下面对于对象的讨论让你感到困惑，请随时回头参考上面包含4个要点的列表。请总是将对象的集合想象成一个在内存中巨大的信息表，表中一行即为对象而列为属性。

8.2 为什么PowerShell使用对象

PowerShell使用对象来代表数据的一个原因是，当然你总需要某种方式代表数据，对吧？**PowerShell**可以将数据以类似XML的格式存储，或使用纯文本表来代表。但微软是有一些具体的理由不这么做。

第一个原因是Windows本身就是一个面向对象的操作系统——或者至少，大部分在Windows上运行的软件是面向对象的。选择将数据构建成为对象集合的方式将非常容易，因为大部分操作系统喜欢这种结构。

另一个使用对象的原因是这样会使事情简单，并给你提供更加强大的功能和更好的灵活性。现在，让我们假装PowerShell并不会生成对象作为命令的输出结果，而是生成一个简单的文本表。这也是你一开始认为的方式。当你运行类似Get-Process的命令时，你将会得到格式化的文本作为输出结果：

PS C:\> get-process

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
-----	-----	-----	-----	-----	-----	--	-----
39	5	1876	4340	52	11.33	1920	conhost
31	4	792	2260	22	0.00	2460	conhost
29	4	828	2284	41	0.25	3192	conhost
574	12	1864	3896	43	1.30	316	csrss
181	13	5892	6348	59	9.14	356	csrss
306	29	13936	18312	139	4.36	1300	dfsrs
125	15	2528	6048	37	0.17	1756	dfsrv
5159	7329	85052	86436	118	1.80	1356	dns

如果我们希望针对上述信息进行一些操作时会怎样？或许你希望针对所有运行Conhost的进程进行操作。为了完成该项操作，你必须对进程列表进行过滤。在Unix或Linux Shell中，你需要使用类似Grep的命令，并告诉该命令“请帮我检查这个文本列表，仅保留第58~64列包含‘conhost’字符的行，并删除其他行”。结果列表将会仅包含你所指定的进程：

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
-----	-----	-----	-----	-----	-----	--	-----
39	5	1876	4340	52	11.33	1920	conhost
31	4	792	2260	22	0.00	2460	conhost

29	4	828	2284	41	0.25	3192	conhost
----	---	-----	------	----	------	------	---------

接下来将上述文本通过管道传递给另一个命令，比如说从列表中获取进程ID。“从第52~56列中获取字符，但丢弃前两列。”结果可能为：

```
1920
2460
3192
```

最终，你将上述文本通过管道传递给另一个命令，使用该命令杀死这些ID所代表的进程（或任何你希望做的操作）。

这实际上也是Unix和Linux管理员的工作。他们花费大量的时间学习如何更好地解析文本，使用类似Grep、Awk和Sed等工具，并必须熟练使用正则表达式。这一系列过程使得他们更容易定义他们希望计算机查找的文本模式。Unix和Linux从业人员喜欢类似Perl的语言，因为该语言包含丰富的文本解析和文本操作方法。

但这种基于文本的方式存在一些问题：

- 你需要花费更多的时间在文本中打转，而不是完成真正的工作。
- 如果命令的输出结果改变——比如说，将ProcessName列移到表的第一列——你需要重写所有的命令，这是因为这些命令需要依赖列位置之类的东西。
- 你需要善于使用解析文本的语言或工具。不仅由于你的工作需要解析文本，解析文本还是实现目的的手段。

PowerShell使用对象消除所有的文本操作开销。由于对象的工作机制类似内存中的表，因此你无须告知PowerShell信息所在的文本位置，而是仅仅需要输入列名。无论在屏幕或文件中如何组织输出结果，PowerShell都知道去哪里获取数据，内存表总是同一个，因此你永远都不需要由于列移动而重写命令。这样的好处是你更多专注于如何实现功能，而不是这类不必要的开销。

当然，你必须学习一些使得你可以构建PowerShell属性的语法，但所需学习的内容将会比那些纯粹基于文本的Shell要少很多。

8.3 探索对象：Get-Member

如果说对象就像内存中一个巨大的表，而PowerShell仅仅在屏幕上展示表的一部分，那么如何看到其他你需要使用的属性呢？此时如果你想到使用Help命令，我们会很欣慰，因为毕竟我们在之前章节不遗余力地推崇使用帮助。但遗憾的是，这并不对。

帮助系统仅记录背景概念（以“关于”帮助主题的形式）和命令语法。如果需要了解更多关于对象的内容，使用另一个命令：Get-Member。你应该习惯于使用该命令。实际上，你更应该了解输入该命令的快捷方式。我们现在就提供给你：别名Gm。

可以在任何产生某些输出的命令之后使用Gm。例如，你已经知道运行Get-Process会在屏幕上产生一些输出，你可以将这些输出通过管道传送给Gm：

```
Get-Process | Gm
```

当一个Cmdlet产生一个对象的集合时，就像Get-Process命令那样，整个集合直到管道末尾之前都可以被访问。直到最后一个命令运行完之前，PowerShell都不会将对象的89个标签或属性过滤掉。直到最后一个命令运行完，才会创建你所见到的文本输出结果。因此在之前的例子中，Gm可以完整访问进程对象的属性和方法，这是由于该命令还未被过滤用于显示。Gm会查看每一个对象并构建一个包含对象属性和方法的列表，该列表内容如下：

```
PS C:\> get-process | gm

      TypeName: System.Diagnostics.Process
Name      MemberType      Definition
----      -
Handles   AliasProperty    Handles = Handlecount
Name      AliasProperty    Name = ProcessName
NPM       AliasProperty    NPM =
NonpagedSystemMemo...
PM        AliasProperty    PM = PagedMemorySize
VM        AliasProperty    VM = VirtualMemorySize
WS        AliasProperty    WS = WorkingSet
Disposed  Event            System.EventHandler
```

Disp...		
ErrorDataReceived	Event	
System.Diagnostics.DataR...		
Exited	Event	System.EventHandler
Exit...		
OutputDataReceived	Event	
System.Diagnostics.DataR...		
BeginErrorReadLine	Method	System.Void
BeginErrorRe...		
BeginOutputReadLine	Method	System.Void BeginOutputR...
CancelErrorRead	Method	System.Void
CancelErrorR...		
CancelOutputRead	Method	System.Void
CancelOutput...		

由于列表过长，我们对上述列表进行了裁剪。但愿你能理解其中的意思。

动手实验： 不要只相信我们所说的。现在你可以趁热打铁运行一些我们提供的命令，以便查看完整的输出结果。

顺便说一下，还有一个可能会让你感兴趣的知识点，就是一个对象的属性、方法以及其他附加到对象的东西都被称为成员。就好像对象本身是一个乡村俱乐部，所有属性和方法都是俱乐部的成员。这也是Get-Member名称的由来：该命令获取对象成员的列表。但请记住，PowerShell中的惯例是使用单数名词，所以Cmdlet的名称为Get-Member，而不是“Get-Members”。

重要： 请注意Get-Member输出结果的第一行，这一行很容易被忽视。这一行是TypeName，是分配给特定类型对象的唯一名称。它现在看起来好像并不重要——毕竟，谁会关心它的名称呢？但该名称将会在下一章节成为关键内容。

8.4 对象标签，也就是所谓的“属性”

当你查看Gm的输出结果时，你会注意到一些不同种类的属性：

- 脚本属性；
- 属性；
- NoteProperty；

- 别名属性。

补充说明

通常来说，.Net Framework中的对象——也就是所有PowerShell对象的来源——只包含“属性”。PowerShell会动态添加其他内容：ScriptProperty、NoteProperty、AliasProperty等。如果你正好在微软的MSDN文档中查看某个对象类型（你可以将对象的类型名称输入MSDN的搜索框），你无法找到这些额外的属性。

PowerShell有一个扩展类型系统（ETS）负责添加这些后来的属性。为什么它会这么做？拿某些案例来说，它使得对象具有更好的一致性，比如为原生只具有类似ProcessName属性的对象添加Name属性（这也是别名属性的作用）。还有一些情况是暴露对象中隐藏的一些信息（进程对象包含一些脚本属性完成这项工作）。

当你在PowerShell的世界中，这些属性的行为都会变得一致。但当这些属性并没有在官方文档页面中出现时，也请不要惊讶：Shell会自动添加这些额外的属性，通常会使得你的工作更加轻松。

对实现你的目标来说，这些属性都一样，唯一的区别是属性原本是如何被创建出来的。但你不必担心这些。对你来说，这些都是“属性”，使用的方法并无不同。

属性总是包含一个值。例如，进程对象的ID属性可能是1234，对象的名称属性的值可能是NotePad。属性用于描述关于对象的某些方面：它的状态、它的ID、它的名称等。在PowerShell中，属性通常是只读的，意味着你无法通过给Name属性赋一个新值来改变服务的名称。但你可以通过读取Name属性来获取服务的名称。我们估计你在PowerShell中90%的工作都需要与属性打交道。

8.5 对象行为，也就是所谓的“方法”

很多对象都支持一个或多个方法，正如我们之前提到过的，是你指导对象的行为。进程对象包含一个Kill方法，它会终止进程。某些方法需要一个或多个输入参数来为某个行为提供额外的细节信息，

但在早期的PowerShell学习中，你不会遇到这些需要参数的方法。实际上，你可能使用多个月甚至多年PowerShell而从来不需要执行一个有参数的方法，这是由于这些方法可以和Cmdlets互相替代。

例如，如果你需要终止进程那个，可以通过三个办法实现。其中一个办法是获取对象并执行Kill方法，另一个办法是使用一系列Cmdlets：

```
Get-Process -Name Notepad | Stop-Process
```

你还可以使用单个Cmdlet完成这项任务：

```
Stop-Process -name Notepad
```

在整本书中，我们更专注于使用PowerShell Cmdlet完成任务。Cmdlet提供了最简单、最管理员导向、最聚焦任务的方式完成工作。而使用方法就开始进入.NET Framework编程的领域，这会更加复杂且需要更多的背景知识。鉴于此，你将会很少—或是从不看到我们在本书中执行对象的方法。实际上，我们在这一点上的哲学是：“如果无法通过Cmdlet完成，那就回头使用GUI完成”。相信我们，在你的职业生涯中都不会感受到这种哲学。但现在来说，保持使用“PowerShell的方式”做事是一个不错的办法。

补充说明

在学习PowerShell的本阶段，你无须懂得关于对象方法的知识。但除了属性和方法之外，对象还有一个事件。事件是以对象的方式通知你某些事情发生了。一个进程对象，举例来说，可以在进程结束时触发Exited事件。你可以将你自己的命令附加到这些事件上，比如说，当进程结束时发送一封邮件。以这种方式 and 事件交互是高级主题，并且超出了本书的范畴。

8.6 排序对象

大部分PowerShell Cmdlets以确定性的方式产生对象，这意味着每次运行命令时都会以相同的顺序产生对象。例如，服务和进程都按照字母表顺序对名称进行排序。事件日志倾向于按照事件排序。那么假如我们希望改变排序方式，该如何做？

例如，假设我们希望显示一个进程列表，按照对虚拟内存（Virtual Memory，VM）的消耗由高到低进行排列。我们将需要基于VM属性对列表进行重新排序。PowerShell提供了一个简单的Cmdlet、Sort-Object，就像其名称那样，可以对对象进行排序：

```
Get-Process | Sort-Object -property VM
```

动手实验： 我们希望运行一些命令。我们不会将输出结果写入书中，因为输出结果表有点长。但如果你跟着教程运行，你会在你的屏幕上得到同样的结果。

该命令并不是我们最终想要的结果。它虽然以VM进行排序，但是以升序形式，最大值在列表底部。通过阅读Sort-Object，可以发现-descending参数可以反转排序。我们还注意到，-property参数是定位参数，因此无须输入参数名称。我们还告诉过你Sort-Object有一个别名，也就是Sort，所以你可以在下一个动手实验中少输入一些内容：

```
Get-Process | Sort VM -desc
```

我们还将-descending简化为-desc，仍然可以得到想要的结果。-property参数接受多个值（如果你查看过帮助文件，我们确定你可以发现这一点）。

为了以防两个进程使用的虚拟内存相同，我们还希望按照进程ID进行排序。下述命令可以实现这一点：

```
Get-Process | Sort VM, ID -desc
```

和之前一样，通过以逗号分隔列表的方式将多个值传递给任意支持多个值的参数。

8.7 选择所需的属性

另一个有用的Cmdlet是Select-Object。该Cmdlet从管道接受对象，你可以指定希望显示的属性。这使得你可以访问任意属性，减少返回列表，只返回你感兴趣的列，而默认情况下由PowerShell配置规则控制。这对于将对象输出到HTML的ConvertTo-HTML命令来说非常有用，因为该Cmdlet通常会创建包含所有属性的表。

比较下面两个命令的结果：

```
Get-Process | ConvertTo-HTML | Out-File test1.html
Get-Process | Select-Object -property Name, ID, VM, PM |
➔ ConvertTo-HTML | Out-File test2.html
```

动手实验： 请尝试分别运行上述命令，然后在IE浏览器中查看输出的HTML结果，以比较区别。

请花一些时间查看Select-Object（或者可以使用该命令别名：Select）。-property参数看上去是定位参数，这意味着我们可以将上面运行的命令缩短：

```
Get-Process | Select Name, ID, VM, PM | ConvertTo-HTML | Out-File
test3.html
```

请花一些时间体验Select-Object。实际上，可以修改下述命令进行其他尝试，该命令将结果展现在屏幕上。

```
Get-Process | Select Name, ID, VM, PM
```

请尝试从列表中添加或删除不同的进程对象属性并查看结果。在最多可以指定多少属性的情况下保持输出结果以表的形式展现？在选

择多少属性的情况下就会强制PowerShell在输出结果中使用别名而不是表？

补充说明

Select-Object还拥有-First和-Last参数，这两个参数可以保留管道中对象的子集。例如，`Get-Process | Select First 10` 将会保留前10个对象。但不能加过滤条件，比如选择特定的进程，只能选择前（或最后）10个。

警告： 人们经常会将Select-Object和Where-Object这两个PowerShell命令搞混，虽然目前你还没有见过Where-Object。Select-Object用于选择所需的属性（或列），还可以选择输出行的任意子集（使用-First和-Last）。Where-object基于筛选条件从管道中移除或过滤对象。

8.8 在命令结束之前总是对象的形式

PowerShell管道在最后一个命令执行之前总是传递对象。在最后一个命令执行时，PowerShell将会查看管道中所包含的对象，并根据不同的配置文件决定哪一个属性被用于构建展示在屏幕上的最终结果。它还会基于一些内部规则和配置文件确定展示是表还是列表（我们将会在接下来的章节阐述更多关于这些规则和配置，以及如何修改它们）。

一个重要的事实是，在一个命令行中，管道可以包含不同类型的对象。在接下来的例子中，我们将会选择一个命令行，并且每一个命令单独占一行，这样将更容易解释我们所谈论的内容。

下面是第一个示例。

```
Get-Process |  
Sort-Object VM -descending |  
Out-File c:\procs.txt
```

在本例中，首先运行Get-Process，该命令将进程对象放入管道。下一个命令是Sort-Object，该命令并不会改变管道中的内容，仅仅是改变对象的顺序，直到Sort-Object结束，管道仍然只包含进程。最后一个命令是Out-File。在这里，PowerShell生成输出结果，也就是管道中所包

含的内容—进程对象，并根据PowerShell的内部规则将对象格式化，最终结果存入指定文件。

接下来是一个稍复杂的例子。

```
Get-Process |  
Sort-Object VM -descending |  
Select-Object Name, ID, VM
```

该命令以同样的方式运行。**Get-Process**将进程对象放入管道。接下来运行**Sort-Object**，该命令将同样的进程对象放入管道。但**Select-Object**就有所不同了。进程对象总是拥有相同的成员。**Select-Object**并不能通过删除你不需要的属性减少属性列表。如果这样的话，结果就不再是进程对象，而是**Select-Object**创建一个名为**PSObject**的自定义对象，PowerShell使用这个对象将属性从进程对象中复制出来，结果是自定义对象被放入管道。

动手实验： 尝试在一个命令行中输入上述3个Cmdlet。请记住，你需要在一行中输入所有的命令。请注意输出结果和正常运行**Get-Process**的输出结果有何不同。

当PowerShell发现光标已经到达命令行结尾时，它必须知道如何对文本输出结果进行排版。这是由于管道中包含的对象不再是进程对象，PowerShell不会再将默认规则和配置应用于进程对象，而是通过查询**PSObject**的规则和配置，这也是当前管道中包含的配置类型。由于**PSObjects**用于自定义输出，微软并没有为**PSObjects**提供任何规则或配置。而是PowerShell将尽最大努力进行猜测并产生表。在理论上，产生的表可以容纳上述3列信息，但表并不像正常的**Get-Process**输出结果那样排版很好看，这是由于Shell缺少使得表更好看的额外的配置信息。

你可以使用**Gm**查看管道中不同的对象。请记住，你可以在任何产生输出结果的Cmdlet之后使用**Gm**。

```
Get-Process | Sort VM -descending | gm  
Get-Process | Sort VM -descending | Select Name, ID, VM | gm
```

动手实验： 请分别运行上述两个命令，并查看输出结果的区别。

请注意，PowerShell会展示出管道中对象的类型名称作为Gm输出结果的一部分。在第一个例子中，对象类型为System.Diagnostics.Process，但是在第二个例子中，管道里包含另一种类型的对象。这个新的“经过筛选”的对象仅包含3个指定属性—Name、ID和VM，以及另外一些由系统生成的成员。

即便Gm产生对象并将对象放入管道，在运行Gm之后，管道也不再包含进程对象或是“经过筛选”的对象，它仅包含由Gm生成的对象类型：Microsoft.PowerShell.Commands.MemberDefinition。你可以通过在管道中对Gm的输出结果再次使用Gm命令证明：

```
Get-Process | Gm | Gm
```

动手实验： 你一定很想尝试该命令，该命令让人感到有些费解。首先是Get-Process命令，将进程对象放入管道。然后运行Gm，该命令分析进程对象并生成该对象的MemberDefinition对象。然后将结果再次利用管道传输给Gm，该命令将分析并产生MemberDefinition成员列表作为输出结果。

掌握PowerShell的一个关键点是在任意时间点知道当前管道中的对象类型。Gm可以帮助你实现这一点，但自己将整个命令从头到尾过一遍将会更好地帮助你理清头绪。

8.9 常见误区

参加我们课程的学生在开始学习PowerShell时通常会犯一些错误，虽然随着经验的积累，这些错误都会被修正，但我们还是希望他们所犯的错误会引起你的警觉。下面的列表可以帮助你走错方向时及时改正。

- 请记住，PowerShell帮助文件不包括有关对象属性的信息。你必须将对象利用管道传输给Gm（Get-Member）来查看属性列表。

- 请记住，你可以在产生结果的任意管道末尾添加Gm命令。类似Get-Process -name Notepad | Stop-Process的命令行正常情况下不产生结果，所以将|Gm置于管道末尾不会产生任何结果。
- 请注意输入的整洁性。请在管道操作符两边加入空格，这是由于命令行看起来更像Get-Process | Gm，而不是Get-Process|Gm。在这里添加空格是有原因的，请使用空格。
- 请记住，管道中在不同阶段可以包含不同类型的对象。请考虑当前在管道中的对象类型是什么，并把精力集中在下一个命令对当前类型的对象所做的操作。

8.10 动手实验

注意：

对于本次动手实验来说，你需要运行PowerShell v3或更新版本PowerShell的计算机。

目前为止，本章或许比其他章节覆盖了更多、更难以及更新的知识。希望我们的讲述方式能够帮你理解这些概念。下面的练习可以帮助你巩固所学到的知识。请尝试完成所有练习，并根据MoreLunches.com的配套视频和示例代码辅助你的学习。其中一部分任务需要你利用在之前章节所学的知识，这是为了帮你巩固之前的知识。

1. 找出生成随机数字的Cmdlet。
2. 找出显示当前时间和日期的Cmdlet。
3. 任务#2的Cmdlet产生的对象类型是什么？（由Cmdlet产生的对象类型名称是什么？）。
4. 使用任务#2中的Cmdlet和Select-Object，仅显示是星期几，示例如下（警告：输出结果将会靠右对齐，请确定PowerShell窗口没有水平滚动条）：

```
DayOfWeek
```

```
-----
```

5. 找出可以显示已安装的补丁（hotfix）的Cmdlet。

6. 使用任务#5的Cmdlet显示已安装的补丁列表，按照安装日期对列表进行排序，并仅仅显示如下几列：安装日期、补丁ID、安装用户。请记住，在命令默认输出显示的列头并不一定是属性的实际名称——你需要查找实际的属性名称来确保这一点。

7. 重复任务#6，但这次按照补丁描述对结果进行排序，并输出描述、补丁ID、安装日期列，最终将结果保存到HTML文件。

8. 从安全事件日志中显示最新的50条列表（如果安全事件列表为空，你也可以使用其他日志，比如系统或应用程序日志）。按照时间升序对日志进行排序，同时也按照索引排序。显示索引、时间以及每条记录的来源。将这些信息存入文本文件（不是HTML文件，而是纯文本文件）。你可以尝试使用Select-Object以及它们的-first或-last参数实现本任务；但请不要这么做，还会有更好的方法。同时目前请避免使用Get-Winevent；可以使用一个更好的Cmdlet完成本任务。

第9章 深入理解管道

此刻，你已经学到如何高效使用PowerShell的管道。这些命令的功能非常强大（比如`Get-Process | Sort VM-desc | ConvertTo-HTML | Out-File process.html`）。如果采用其他脚本语言实现相同功能，可能需要编写多行代码，但是利用PowerShell，仅需要单行命令即可。但是，你本可以做得更好。在本章中，我们会更深入地讲解管道相关的知识，并展示其更加强大的功能。

9.1 管道：更少的输入，更强大的功能

我们喜欢PowerShell的一个重要原因是它并不像VBScript那样，需要我们写很多的代码来实现某些功能，从而使得我们的工作更加高效。单行的PowerShell命令功能如此强大，主要在于PowerShell管道的工作机制。

另外需要说明：你完全可以跳过本章的学习，也可以高效使用PowerShell。但是在大部分情况下，你不得不采用VBScript的风格编写脚本或者程序。虽然PowerShell的管道功能非常复杂，但是相比于其他更为复杂的编程语言，它更容易学习。通过学习如何使用管道，你可以更高效地完成某项工作，而无须编写脚本。

本章的宗旨是在尽量键入较少的命令的前提下，如何让Shell完成更多的工作。可以想象，你会很惊讶地发现，该Shell可以非常完美地实现这些功能。

9.2 PowerShell如何传输数据给管道

当将两条命令串联在一起时，PowerShell必须搞清楚怎样将第一条命令的输出作为第二条命令的输入。在下面的示例中，我们将第一条命令称为命令A，这条命令会产生某些结果。第二条命令称为命令B，它会接收命令A产生的结果集，然后完成自己的工作。

```
PS C:\>CommandA | CommandB
```

如图9.1所示，在该文本文件中，每行均代表一个计算机的名称。



图9.1 创建一个包含计算机名称的文本文件，每行代表一个计算机名称

你可能希望将这部分计算机名称作为某些命令的传入数据，以便该命令会在这些计算机上被运行，比如下面的例子。

```
PS C:\> Get-Content .\computers.txt | Get-Service
```

当运行**Get-Content**命令时，它会将文本文件中的计算机名称放入管道中。之后PowerShell再决定如何将该数据传递给**Get-Service**命令。但PowerShell一次只能使用单个参数来接收传入数据。也就是说，PowerShell必须决定由**Get-Service**的哪个参数来接收**Get-Content**的输出结果。这个决定的过程就称为管道参数绑定（Pipeline parameter binding），这也是本章主要讲解的内容。PowerShell将使用两种方法来将**Get-Content**的输出结果传入给**Get-Service**的某个参数。该Shell尝试使用的第一种方法称为**ByValue**；如果这种方法行不通，它将会尝试**ByPropertyName**。

9.3 方案A：使用ByValue进行管道输入

当使用**ByValue**这种方式实现管道参数绑定时，PowerShell会确认命令A产生的数据对象类型，然后查看命令B中哪个参数可以接受经由管道传来对象的类型。可以采用下面的方法来证明：通过管道将命令A的输出结果发送给**Get-Member**，然后就可以查到该命令产生的结果的对象类型。之后，查

看命令B的详细帮助信息（例如Help Get-Service -Full），确定命令B的哪个参数可以接收ByValue管道传出的数据类型。图9.2展示了该过程。

你将会看到Get-Content命令产生的结果对象的类型是System.String（或者简称为String）。通过查询帮助信息，可以看到Get-Service中的确也存在可以从ByValue管道中接收String类型数据的参数。检查发现，可以接受String类型数据的参数是-Name。查看帮助信息，其说明为“指定要检索的服务的名称”。你可能已经发现一个问题：这并不是我们需要的——我们的文本文件中的内容，也就是String对象，是指计算机名称，并不是服务的名称。如果我们执行下面的命令，之后会得到名为SERVER2或者WIN8的服务名，肯定无法正常执行。

```
PS C:\> Get-Content .\computers.txt | Get-Service
```

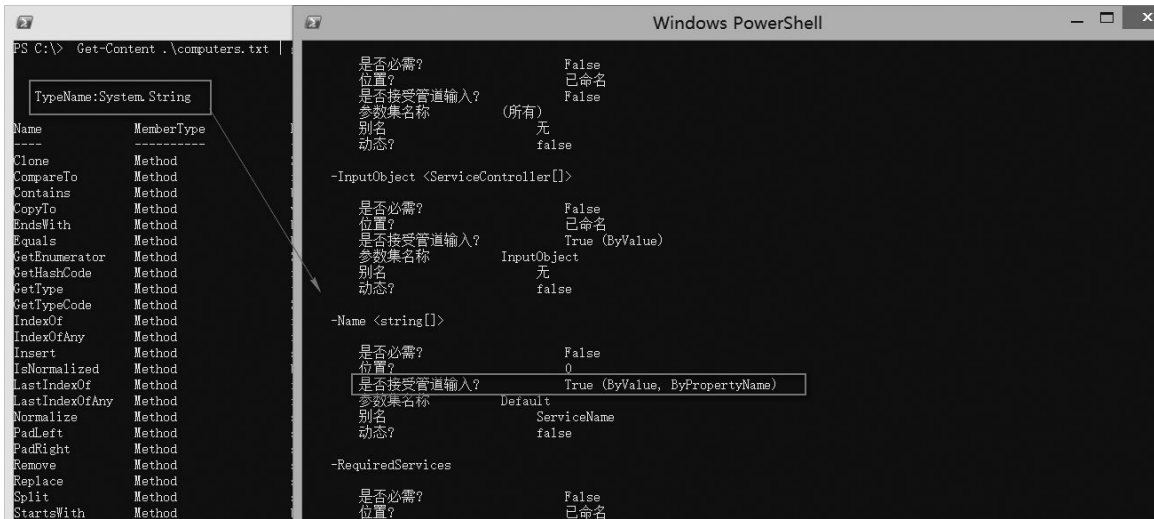


图9.2 对比Get-Content的输出结果与Get-Service的输入参数

PowerShell只允许使用一个参数去接收ByValue管道返回的对象类型。也就意味着，由于-Name参数接收了来自ByValue管道返回的String类型数据，那么其他参数就无法接收该数据了。这样我们将文本文件中的计算机名称通过管道传递给Get-Service命令的希望破灭了。

在这个示例中，管道的输入可以正常工作，但是无法得到我们期望的结果。我们再看另外一个示例。在新示例中，我们能得到我们期望的结果。下面是对应的命令行：


```
PS C:\>Get-Process -Name note* | Stop-Process
```

我们将命令A的输出结果通过管道传递给Get-Member，之后查看命令B的详细帮助信息。图9.3即为之后的对比结果。

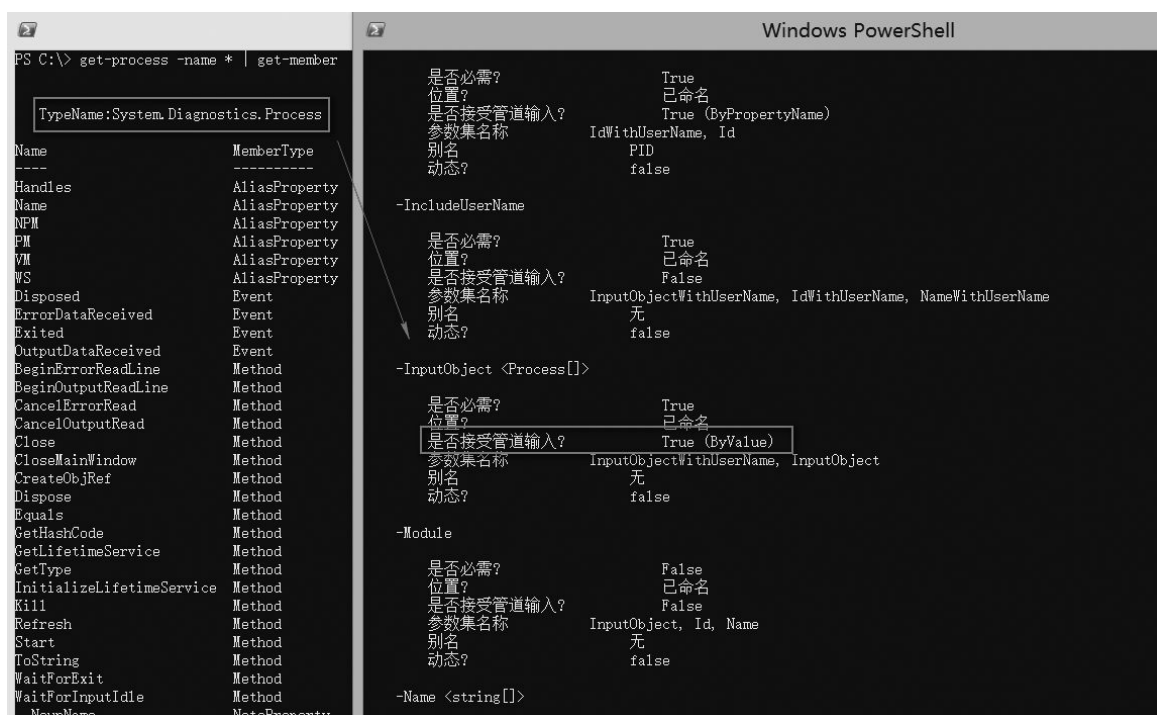


图9.3 将Get-Process输出结果绑定到Stop-Service命令的一个参数

Get-Process命令会返回类型为System.Diagnostics.Process的对象（注意：我们在该示例中限制了返回的Process的名称（名称以note开头）；由于我们开启一个NotePad进程，所以执行该命令后，会返回对应的结果）。Stop-Process命令会使用-InputObject参数接收这些来自ByValue管道的进程对象。从帮助信息中得知，该参数会“停止由指定的进程对象表示的进程”。换句话说，命令A会返回一个或多个进程对象，命令B会停止（或者杀死）这些进程。

这是诠释管道参数绑定一个比较恰当的示例，同时反映了PowerShell中比较重要的一个知识点：大部分情况下，使用相同名词的命令都可以使用ByValue方式相互之间进行管道传输（比如Get-Process和Stop-Process）。

下面，我们看另外一个示例：

```
PS C:\>Get-Service -Name s* | Stop-Process
```

表面上看起来，这个命令没有任何意义。但是当我们把命令A的结果集通过管道传输给Get-Member，之后再查看命令B的详细帮助信息，那么也就如图9.4所示。

Get-Service返回了ServiceController类型的对象（准确地说，应该是System.ServiceProcess.ServiceController，但是我们可以只取最后一位的名称作为简写）。糟糕的是，Stop-Process没有一个参数可以接收ServiceController类型的对象。也就意味着，使用ByValue方式进行处理方案失败，此时PowerShell会尝试其备选方案ByPropertyName。

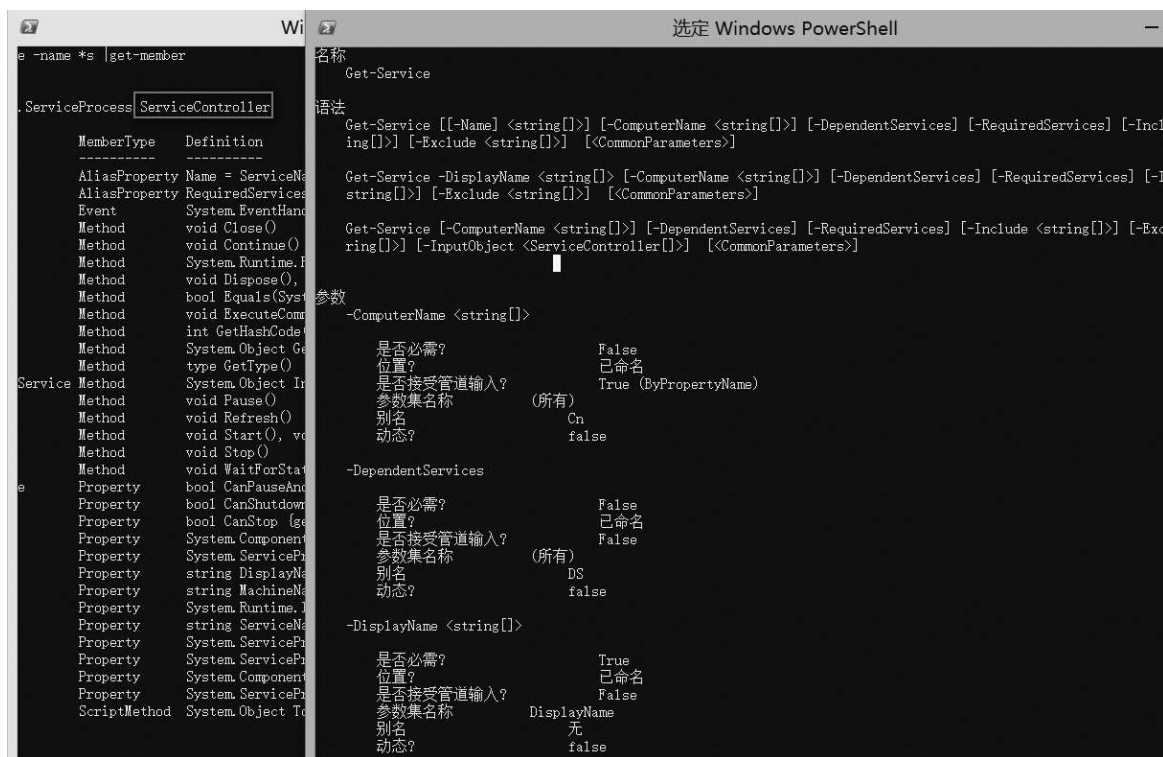


图9.4 检查Get-Process的输出结果以及Stop-Process的输入参数

9.4 方案B：使用ByPropertyName进行管道传输

该方案同样需要将命令A的输出结果传递给命令B的参数。但是ByPropertyName与ByValue稍有不同。通过该方法，命令B的多个参数可以被同时使用。我们再次将命令A的输出结果传递给Get-Member，之后查看

命令B的语法。图9.5展示了该结果：命令A的输出结果中一个属性的名称匹配到命令B的一个参数。

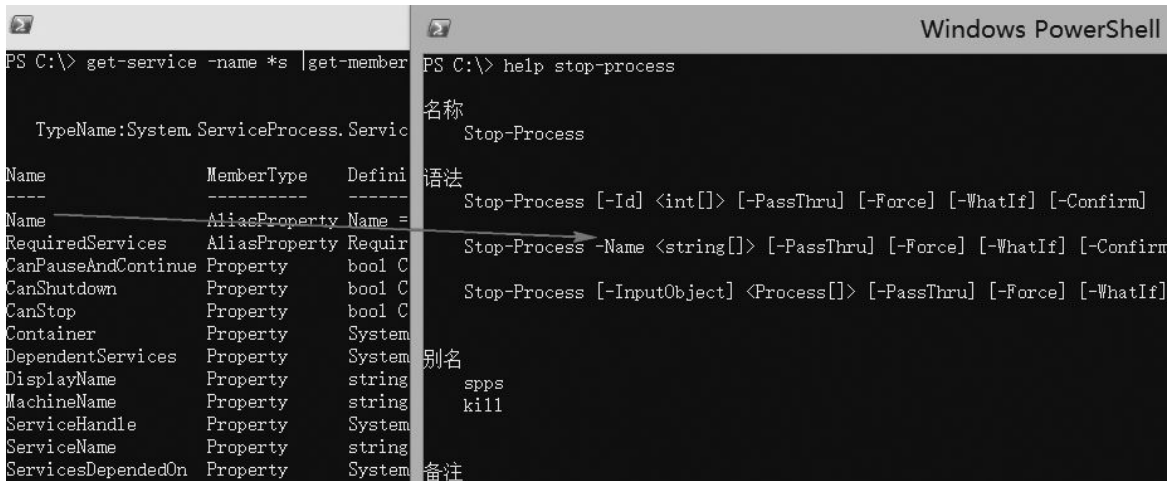


图9.5 映射属性到参数

很多人都会认为这里的原理很复杂，因此需要澄清一下，该Shell对该功能的实现其实非常简单：仅仅是寻找能够匹配参数名称的属性名称。就是这么简单，本例中属性“Name”与参数名称“-Name”相同，Shell会尝试将这两个值进行关联。

但是并不是如此简单就能实现：首先，它会检查-Name参数是否可以接收来自ByPropertyName管道的输出。通过查看详细帮助信息，就可以确定，如图9.6所示。

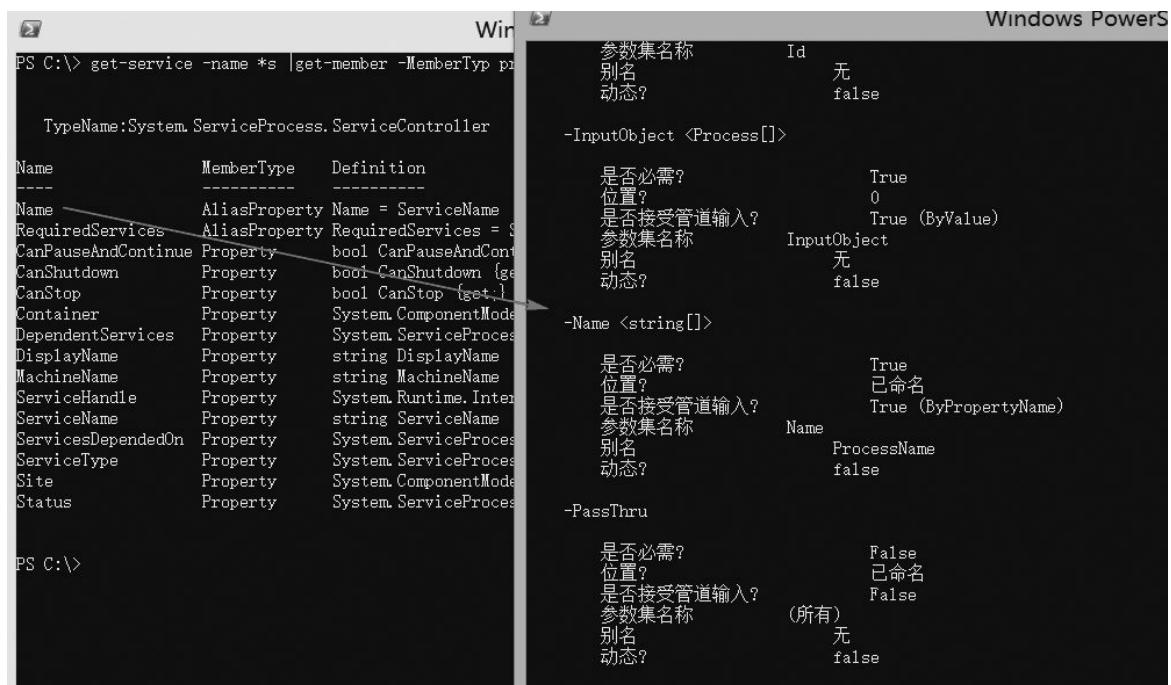
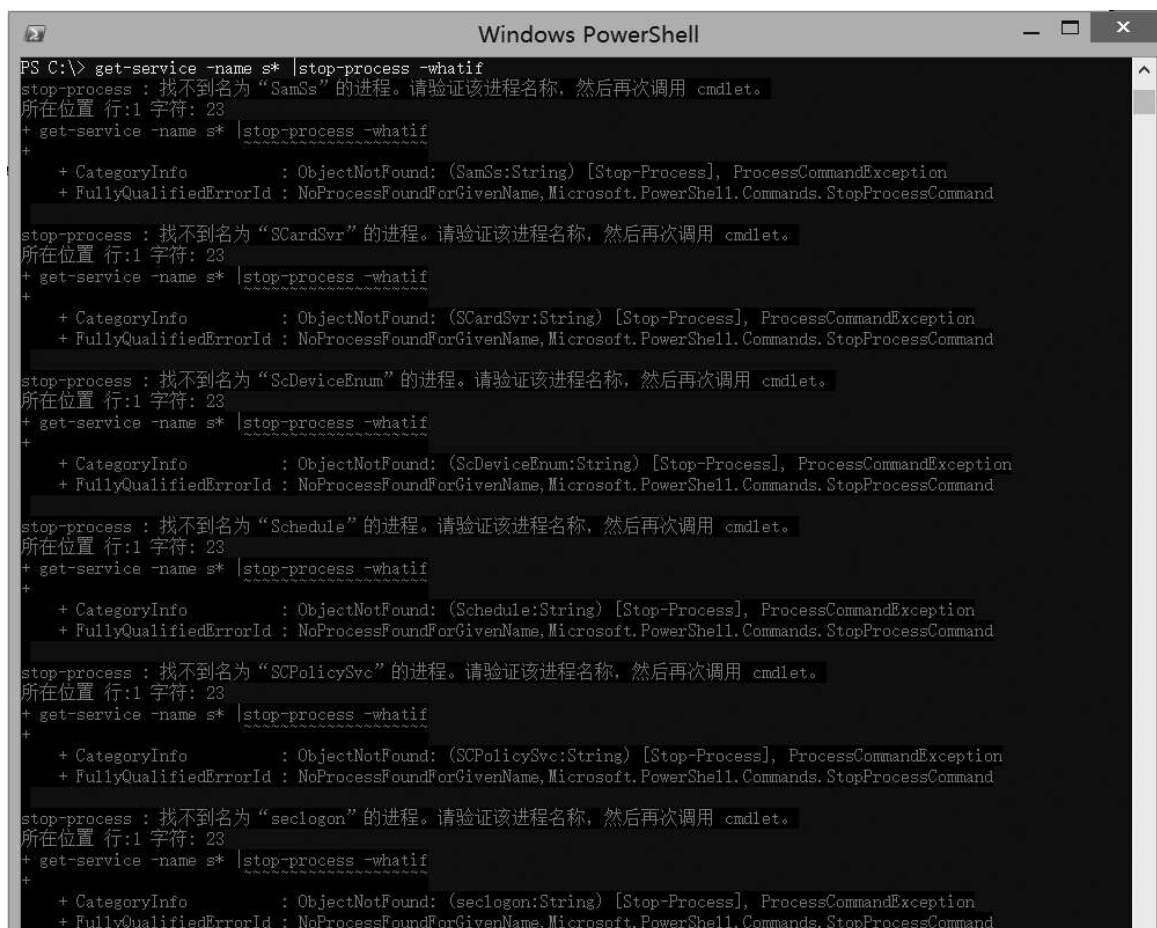


图9.6 确认Stop-Process的-Name参数是否可以接收ByPropertyName管道输出结果

在这个示例中，`-Name`参数可以接收来自`ByPropertyName`管道的输出结果，所以这个连接可以正常工作。神奇之处在于，与`ByValue`管道只能使用一个参数不同，`ByPropertyName`会将每个匹配的属性与参数进行关联（提供的每个参数都可以接收来自`ByPropertyName`管道的输出值）。在这个示例中，只有`Name`属性与`-Name`参数匹配，如图9.7所示。

从图9.7中可以看到产生了大量的错误。问题在于，`Service`的名称基本上都类似于`ShellHWDetection`和`SessionEnv`，但是服务的可执行文件一般为类似`svchost.exe`的这种命名规则。`Stop-Process`只会处理那些可执行文件的名字。虽然`Name`属性能通过管道关联到`-Name`参数，但是`Name`属性中隐藏的属性值并不能被`-Name`参数所处理，最终也就导致了上面的错误。



```
PS C:\> get-service -name s* | stop-process -whatif
stop-process : 找不到名为“SamSs”的进程。请验证该进程名称，然后再次调用 cmdlet。
所在位置 行:1 字符: 23
+ get-service -name s* | stop-process -whatif
+ ~~~~~
+ CategoryInfo          : ObjectNotFound: (SamSs:String) [Stop-Process], ProcessCommandException
+ FullyQualifiedErrorId : NoProcessFoundForGivenName,Microsoft.PowerShell.Commands.StopProcessCommand

stop-process : 找不到名为“SCardSvr”的进程。请验证该进程名称，然后再次调用 cmdlet。
所在位置 行:1 字符: 23
+ get-service -name s* | stop-process -whatif
+ ~~~~~
+ CategoryInfo          : ObjectNotFound: (SCardSvr:String) [Stop-Process], ProcessCommandException
+ FullyQualifiedErrorId : NoProcessFoundForGivenName,Microsoft.PowerShell.Commands.StopProcessCommand

stop-process : 找不到名为“ScDeviceEnum”的进程。请验证该进程名称，然后再次调用 cmdlet。
所在位置 行:1 字符: 23
+ get-service -name s* | stop-process -whatif
+ ~~~~~
+ CategoryInfo          : ObjectNotFound: (ScDeviceEnum:String) [Stop-Process], ProcessCommandException
+ FullyQualifiedErrorId : NoProcessFoundForGivenName,Microsoft.PowerShell.Commands.StopProcessCommand

stop-process : 找不到名为“Schedule”的进程。请验证该进程名称，然后再次调用 cmdlet。
所在位置 行:1 字符: 23
+ get-service -name s* | stop-process -whatif
+ ~~~~~
+ CategoryInfo          : ObjectNotFound: (Schedule:String) [Stop-Process], ProcessCommandException
+ FullyQualifiedErrorId : NoProcessFoundForGivenName,Microsoft.PowerShell.Commands.StopProcessCommand

stop-process : 找不到名为“SCPolicySvc”的进程。请验证该进程名称，然后再次调用 cmdlet。
所在位置 行:1 字符: 23
+ get-service -name s* | stop-process -whatif
+ ~~~~~
+ CategoryInfo          : ObjectNotFound: (SCPolicySvc:String) [Stop-Process], ProcessCommandException
+ FullyQualifiedErrorId : NoProcessFoundForGivenName,Microsoft.PowerShell.Commands.StopProcessCommand

stop-process : 找不到名为“seclogon”的进程。请验证该进程名称，然后再次调用 cmdlet。
所在位置 行:1 字符: 23
+ get-service -name s* | stop-process -whatif
+ ~~~~~
+ CategoryInfo          : ObjectNotFound: (seclogon:String) [Stop-Process], ProcessCommandException
+ FullyQualifiedErrorId : NoProcessFoundForGivenName,Microsoft.PowerShell.Commands.StopProcessCommand
```

图9.7 尝试将Get-Service的输出结果通过管道传送给Stop-Process

下面看一个可以正常运行的示例：使用记事本新建一个以逗号间隔的CSV文件，如图9.8所示。

将该文件保存为Alias.CSV，之后回到Shell界面，尝试导入该文件，如图9.9所示。当然，你也可以将Import-CSV的输出结果通过管道传递给Get-Member，这样就可以查看输出的内容。



图9.8 在Windows记事本中新建CSV文件

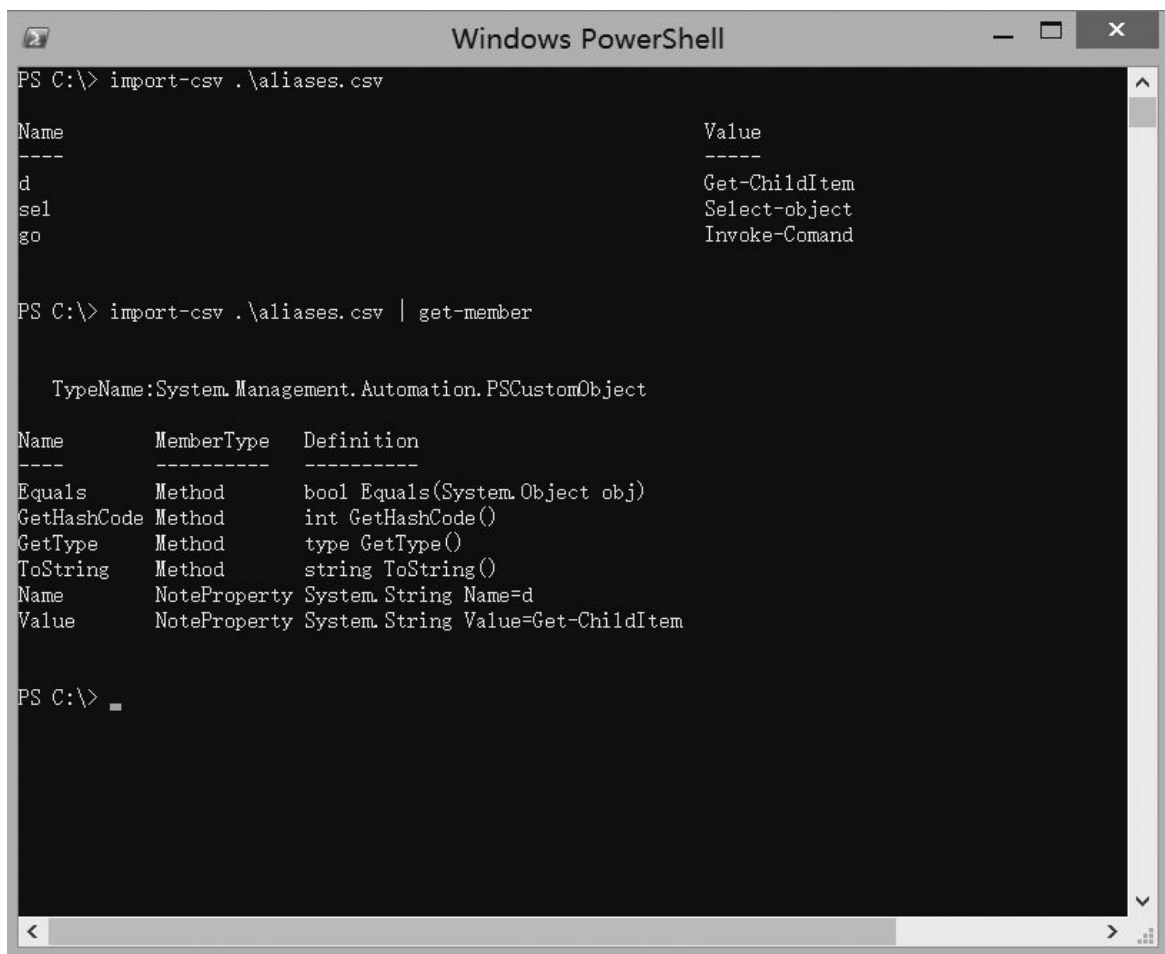


图9.9 导入CSV文件，并查看它的成员

你可以清晰地看到，CSV文件中的列名成了属性，而CSV中每一行的值成了一个对象。现在我们查看New-Alias的详细帮助，如图9.10所示。

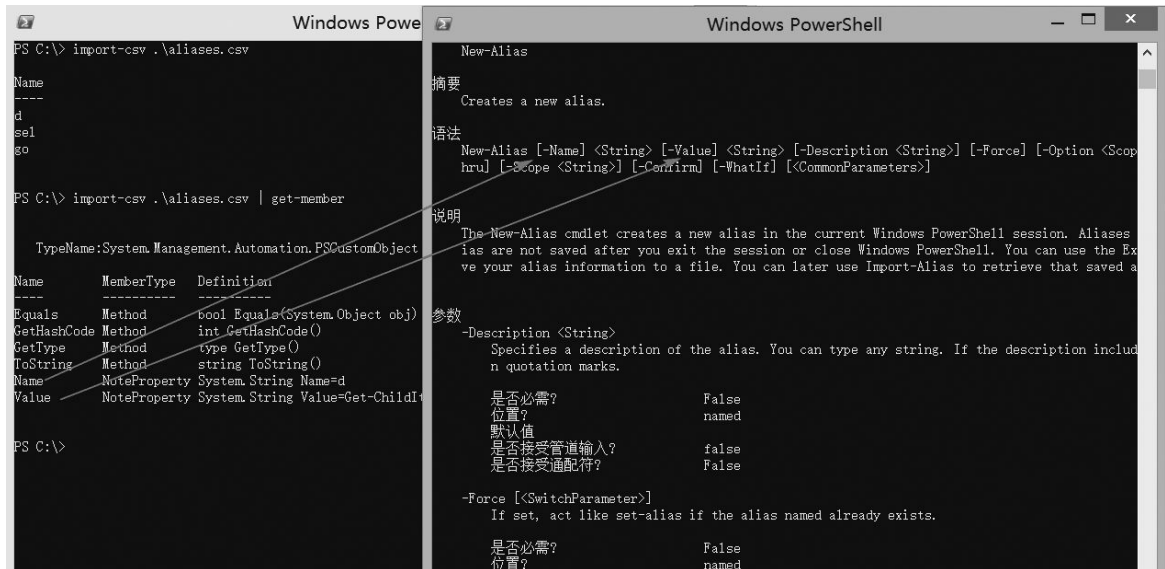


图9.10 匹配属性与对应的参数

Name和Value属性都可以关联到New-Alias的参数名称。当然，这里是特意实现的（因为你可以将CSV文件的列任意命名）。现在我们可以检查New-Alias的-Name和-Value参数是否可以接收来自ByPropertyName管道的输出结果，如图9.11所示。

经过查看，两个参数都可以接收，也就证明下面的语句可以正常工作。尝试执行下面的语句。

```
PS C:\> Import-CSV .\aliases.csv | New-Alias
```

执行之后，会产生三个新的别名，名为d、sel和go，分别对应Get-ChildItem、Select-Object和Invoke-Command命令。从这里可以看出，这是一个非常强大的功能，它可以将数据从一个命令传递给另外一个命令，之后只需要使用少量的命令语句就可以实现复杂的功能。



图9.11 寻找能接受ByPropertyName管道输入的参数

9.5 数据不对齐时：自定义属性

当我们人为创建某些输入数据时，使用CSV是非常简单的一个场景，因为我们可以人为将属性和参数名称对齐。但是当你必须通过PowerShell处理其他对象或者他人提供的的数据时，可能就会变得比较困难。

比如这个示例：我们会介绍一个之前未使用过的命令New-ADUser。该命令属于活动目录中的一个模块，它存在于Windows Server 2008 R2及之后的版本操作系统的域控制器中。另外，你也可以在安装了微软的远程服务器管理工具（Remote Server Administration Tools, RSAT）的客户端电脑上找到该组件。现在请不要担心如何去运行命令，只需要跟随下面的示例就可以了。

New-ADUser命令包含大量参数，每个参数用来匹配一个新的活动目录账号的信息，比如：

- -Name（该参数必须存在）
- -samAccountName（从语法角度，可以不提供。但是仍然需要提供该参数，使得AD账号可用）
- -Department
- -City
- -Title

我们这里本可介绍更多参数，但是如果仅为练习，上面这些参数已经足够。这些参数都可以按照ByPropertyName方式接收管道的输出。

比如下面的例子，你需要处理一个CSV文件，但是该文件来自于公司的HR部门。你可能多次要求他们按照某特定格式给出文件，但是HR部门仍然固执地使用自己的文件格式，如图9.12所示。

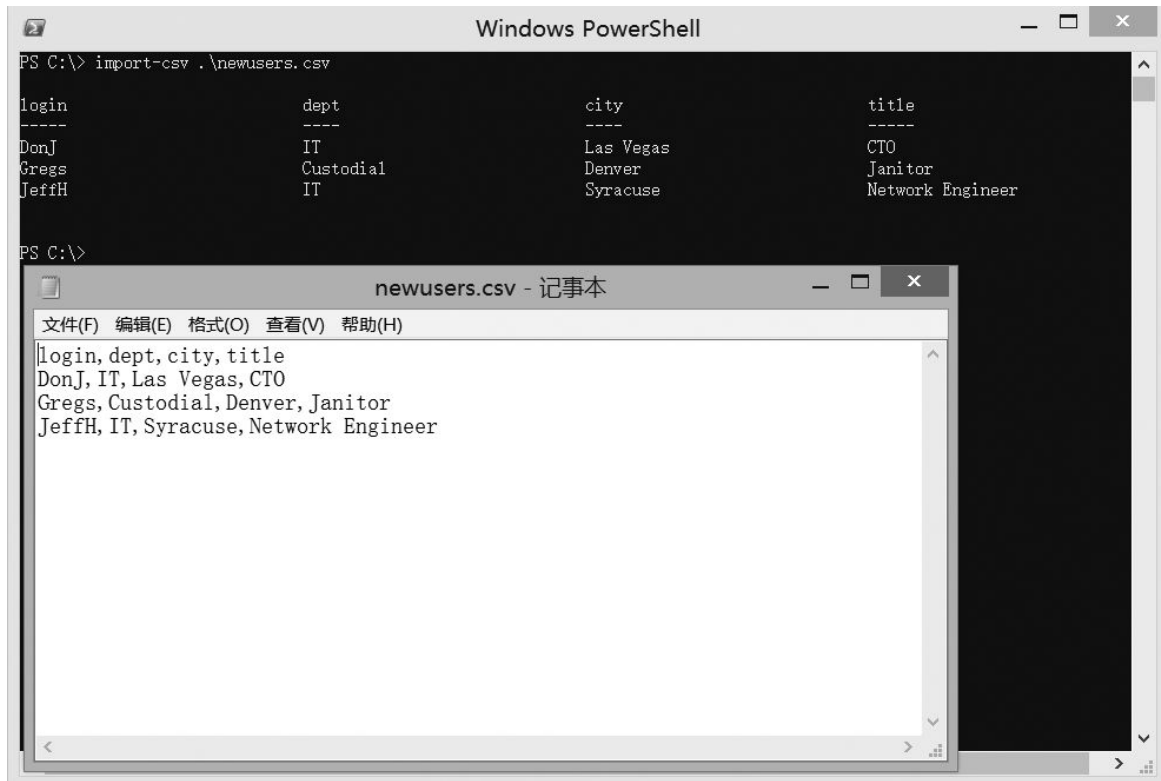


图9.12 处理HR部门提供的CSV文件

如图9.12所示，PowerShell成功导入该CSV文件，最终产生了三个对象，并且每个对象包含四个属性。但是存在一个问题：dept属性与New-ADUser的-Department参数并不吻合；同时Login属性是无意义的，这里并没有包含samAccountName或者Name属性（如果你想通过下面的命令来创建新用户，那么必须指定这两个属性）。

```
PS C:\>Import-Csv .\NewUsers.CSV | New-AdUser
```

那么我们如何解决这个问题？当然，你可以直接打开这个CSV文件，之后修复它（将列名修改为符合New-ADUser中参数的名称），但是需要花

费一定的时间去完成。PowerShell的宗旨在于减少手工劳动。为什么不通过Shell脚本来解决该问题？ 来看下面的示例：

```
PS C:\> Import-CSV .\NewUsers.CSV |
>> Select-Object -Property *,
>> @{name='samAccountName';expression={$_.login}},
>> @{label='Name';expression={$_.login}},
>> @{n='Department';e={$_.Dept}}
>>

Login           : DonJ
Dept            : IT
City            : Las Vegas
Title           : CTO
SamAccountName  : DonJ
Name            : DonJ
Department      : IT

Login           : GregS
Dept            : Custodial
City            : Denver
Title           : Janitor
samAccountName  : GregS
Name            : GregS
Department      : Custodial

Login           : JeffH
Dept            : IT
City            : Syracuse
Title           : NetWork Engineer
SamAccountName : JeffH
Name            : JeffH
Department      : IT
```

看起来，语法比较特别。下面将这部分语法拆开来看：

- 这里我们使用了**Select-Object**命令以及它的**-Property**参数。最开始，我们指定了*这个属性（*是指“所有存在的属性”）。在*后面，我们使用了逗号，也就意味着我们还会输入其他的一些属性列。
- 之后我们创建一个哈希表，哈希表的结构是以@{为起始，以}为结尾。哈希表中包含了一个或者多个成对的键-值（**Key-Value**）数据。我们使用**Select-Object**去寻找我们指定的一些特定键。
- **Select-Object**需要寻找的第一个键可以是**Name**、**N**、**Label**或者**L**，该键对应的值也就是我们想创建的属性的名称。在第一个哈希表中，我们指定了**samAccountName**，第二个哈希表中为**Name**，第三个哈希表中指

定为Department。这三个属性的名称正好可以对应到New-ADUser命令的三个参数。

- **Select-Object**需要的第二个键可以是expression或者E。该键对应的值是一个包含在大括号{}中的脚本块。在脚本块中，使用特定的\$_占位符关联到已存在的管道对象（CSV文件中每行的数据）。通过\$_可以读取管道对象的属性，或者说是CSV文件的一个列。也就是说，通过这种方法来指定新属性的值。

动手实验： 请参照图9.12新建一个CSV文件，之后输入上面示例中运行的所有命令。

到现在为止，已完成的步骤包括获取CSV文件的内容（Import-CSV的输出结果），之后在管道中动态地修改该内容。最后新的数据输出结构能与New-ADUser命令期望的格式一致，这样我们就可以使用下面的命令来创建新的AD用户了。

```
PS C:\> Import-CSV .\NewUsers.CSV |
>> Select-Object -Property *,
>> @{name='samAccountName';expression={$_.login}},
>> @{label='Name';expression={$_.login}},
>> @{n='Department';e={$_.Dept}} |
>> New-ADUser
>>
```

从语法上看，可能不是那么友好，但是确实是一门功能非常强大的技术。在PowerShell的其他地方也可以使用该命令，后续章节中会有类似示例。甚至你可以在PowerShell的帮助文件的示例中看到这种命令：执行Help Select -Example命令就可以发现，但是需要自行查看。

9.6 括号命令

有些时候，不管我们怎么尝试，都无法处理管道的输出结果，比如Get-WMIObject。下一章节中会详细讲解该命令，但是我们现在可以先大概看一下它的帮助信息，如图9.13所示。

该参数并不能接收来自管道的计算机名称。那么我们应该如何将其他来源的数据（比如一个文本文件，其中每行数据代表一个计算机名称）传递给该命令呢？如果按照下面这样编写命令，那么是不能正常执行的。

```
PS C:\>Get-Content .\computers.txt |Get-WMIObject -Class win32_bios
```

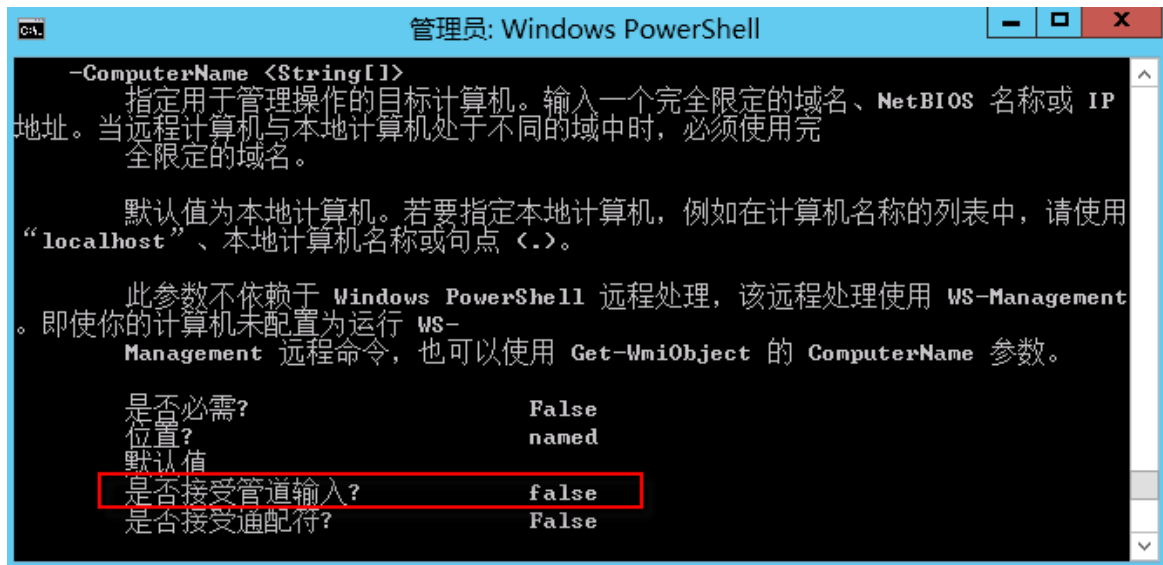


图9.13 查看Get-WMIObject的详细帮助信息

Get-Content命令输出的String对象无法匹配到Get-WMIObject命令的-ComputerName参数。那么此时，我们应该怎么做？使用圆括号。

```
PS C:\> Get-WMIObject -Class Win32_BIOS -ComputerName  
(Get-Content .\computers.txt)
```

现在我们回想一下高中代数课中对括号的解释：“优先执行”。也就是说，PowerShell会采用如下顺序来执行这个命令：先执行括号里的命令；第一步命令执行的结果（在本例中，是多个String类型的对象）被传递给Get-WMIObject的参数。由于-ComputerName能够接收String类型的对象，所以此时，整个命令可以正常执行。

动手实验：如果有大量的计算机可以用来做测试，那样最好不过了。将正确的机器名和IP地址写入到一个computers.txt文件中。如果是在域环境中（在域环境中，计算机的权限变更会非常容易），那么会测试得更顺利。

括号命令功能非常强大，因为它根本不依赖于参数管道绑定——它会将获取的对象强制匹配到正确的参数。但是如果括号中输出的对象类型和

需要绑定的参数类型不一致，也会存在问题。此时，我们需要手动做一些修改。详见下一小节。

9.7 提取属性的值

在本章开始展示了一个示例，在该示例中，我们使用圆括号得到Get-Content的输出结果，之后将该输出结果传递给另外一个Cmdlet的参数。

```
Get-Service -computerName (Get-Content names.txt)
```

在很多时候，我们可能不会从一个静态文件中获取计算机名称，比如可能从活动目录中获取某些数据。借助于ActiveDirectory模块（在Windows Server 2008 R2及之后版本操作系统上，以及在安装了远程服务器管理工具RSAT的客户端电脑上），我们可以查询域控制服务器（Domain Controller）上所有的信息。

```
PS C:\>Get-ADComputer -Filter * -SearchBase "ou=domain controllers,  
➔dc=company,dc=pri"
```

你可以使用括号将上面命令的输出结果传递给Get-Service吗？也就是说，下面的命令可以执行吗？

```
PS C:\>Get-Service -computerName (Get-ADComputer -filter *  
➔-searchBase "ou=domain controllers,dc=company,dc=pri")
```

补充说明

如果你没有域控制器环境，那么也没问题。我们会告诉你需要了解Get-ADComputer的哪些信息。

首先，该命令包含在一个名为ActiveDirectory的模块中。正如前文提到的，在Windows 2008 Server R2以及之后版本操作系统的域控制服务器上，或者在域中某一台已经安装RSAT的客户端计算机上都存在该模块。

其次，正如你猜测的那样，该命令会获取域中的计算机对象。

再次，该命令包含两个非常有用的参数。**-Filter ***将会去到所有计算机上获取对应信息。当然，你也可以指定其他筛选条件来限制返回的结果（比如指定一个特定的计算机名称）。**-SearchBase**参数会告诉这个命令从哪个地方开始查找计算机。在上面的示例中，我们设定该命令从**Company.com**域的域控制器开始查找。

```
Get-ADComputer -Filter * -SearchBase "ou=domain  
➔controllers,dc=company,dc=pri"
```

最后，计算机对象中包含**Name**这个属性，也就是计算机的名称。

我们意识到，直接将这类命令（非常依赖于实验环境）教给你，你可能没法进行测试。从某种程度上说，对你来说可能不太公平。但是在生产环境中，如果真正遇到我们假设的这种场景，该命令会非常有用。如果你能记住前面讲的四点，本节的知识对你将会非常有帮助。

很遗憾，上面的命令无法成功执行。查看**Get-Service**的帮助文件，你可以看到**-Computer**这个参数只能接收**String**类型的值。

请运行下面的命令：

```
Get-ADComputer -Filter * -SearchBase "ou=domain controllers,  
➔dc=company, dc=pri" | gm
```

通过**Get-Member**命令，我们可以看到**Get-ADComputer**命令的输出结果是**ADComputer**类型的对象，而不是**String**类型的对象。所以**-ComputerName**这个参数不知道该如何来处理这部分数据。但是**ADComputer**类型的对象包含了一个**-Name**的属性。接下来我们要做的是，提取出**ADComputer**类型对象中的**-Name**属性值，然后将这些值（也就是计算机名称）传递给**-ComputerName**这个参数。

提示： 这是PowerShell中很重要的一个知识点。如果你还感到不理解或者困惑，那么请停下来重新阅读前文。我们可以通过**Get-Member**命令来确认**Get-ADComputer**命令输出的是**ADComputer**类型的对象；但是查看帮助文档，**-ComputerName**这个参数只能接收**String**类型的对象，而无法处理**ADComputer**类型对象。因此，前面那个包含括号的命令无法正常执行。

再次提醒，我们可以使用**Select-Object**命令来解决这个问题，因为它包含一个可以接收属性名称的参数**-ExpandProperty**。它会获取对应的属性，提

取属性的值，然后返回这些值（作为Select-Object的输出结果）。参考下面这个命令：

```
Get-ADComputer -Filter * -SearchBase "ou=domain controllers,  
→dc=company, dc=pri" | Select-Object -expand name
```

该命令会返回一个包含计算机名称的清单，里面的值可以传递给Get-Service命令的-ComputerName参数（或者其他包含-ComputerName参数的一些Cmdlet）。

```
Get-Service -ComputerName (Get-ADComputer -Filter *  
→-SearchBase "ou=domain controllers,dc=company,dc=pri"|  
→Select-Object -Expand name)
```

提示： 再次申明，这是一个非常重要的概念。一般情形下，类似Select-Object -Property Name这种命令只会返回一个Name的属性（因为我们只指定了该名称）。-ComputerName参数不期望得到任意的带有-Name属性的对象；它更期望得到一个String类型的对象，因为这样会更加简单。-ExpandName会获取Name属性，并且提取其值，最终该命令会输出一些比较简单的String对象。

最后说明一下，这是一个非常棒的技巧，可以将多种命令相互关联。这样可以避免不必要的输入，使得PowerShell可以实现更多的功能。

既然你已经看到使用Get-ADComputer的一些强大功能，下面看另外一个你可以完成的示例。假定你运行的是新版本的操作系统，在这个示例中，不需要计算机在域中，也不需要能访问到域控制服务器，甚至不需要服务器版的操作系统。我们要求得到计算机名称，因为该命令在所有命令中比较常见。

首先，在记事本中创建一个CSV文件，如图9.14所示。如果你在CSV文件中指定的计算机都可以被访问到，那么就可以正常运行示例中的命令。当然，如果只能访问到本机，在HostName列全部写为LocalHost，然后在记事本中写上3次或者4次，最后也可以正常执行该命令。

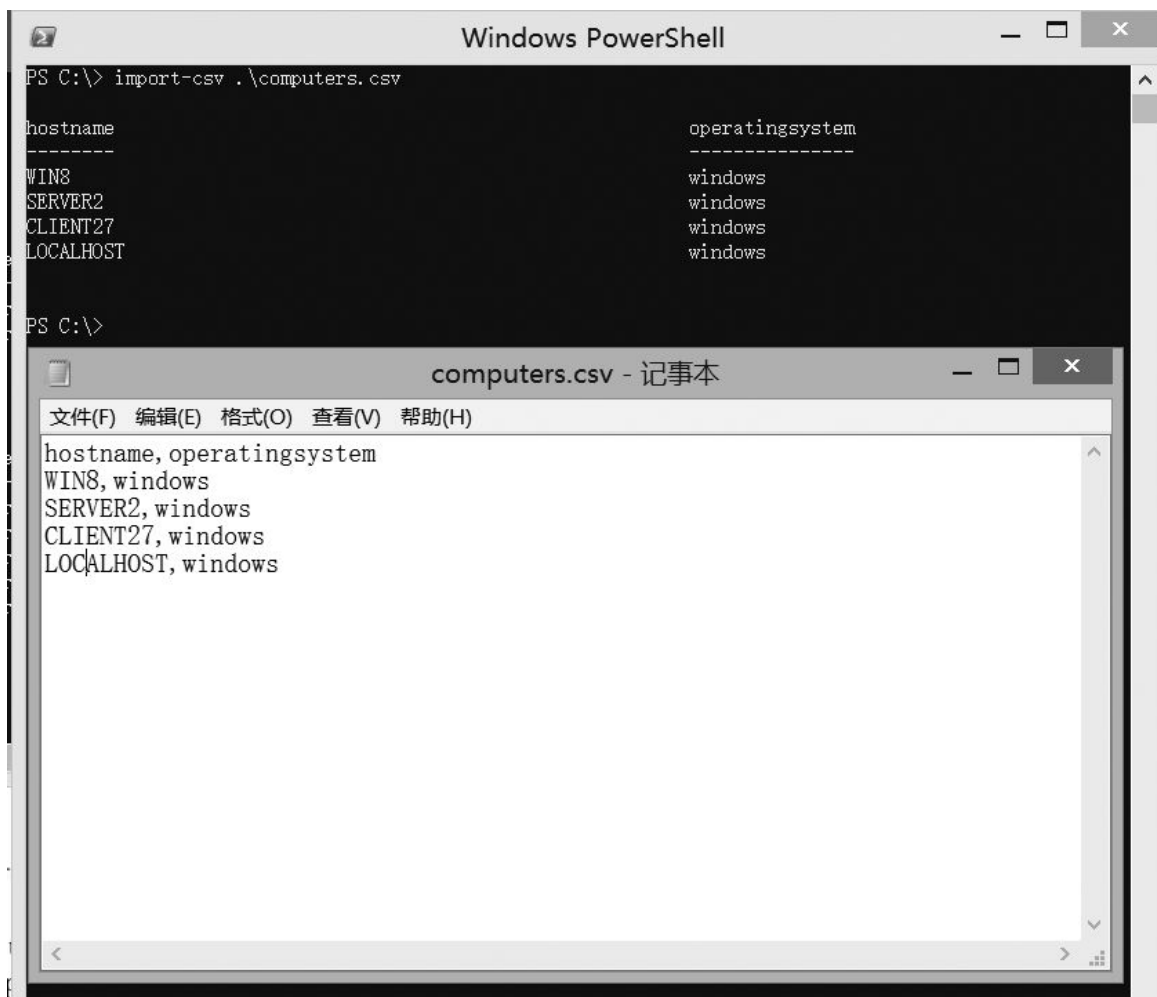
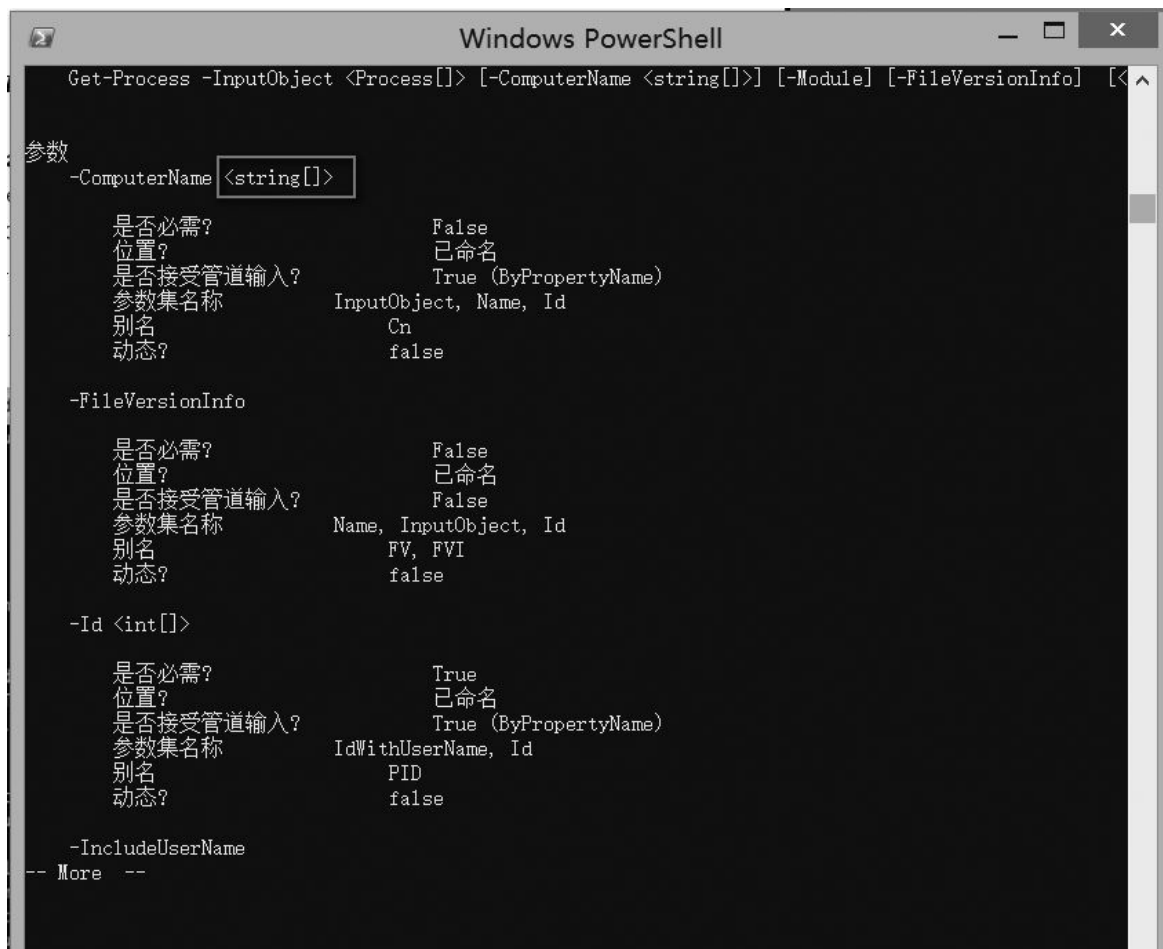


图9.14 确定可以使用Import-CSV导入该CSV文件，得出如图的类似结果

现在我们可以从列出的这部分计算机上找到正在运行的进程列表。通过查看Get-Process命令的帮助文件，你会发现它的-ComputerName参数可以接收ByPropertyName管道的输入。可接收的对象类型为String，这里我们不会关注管道输入。相反，我们关注属性的提取操作。帮助文件中显示-ComputerName参数需要String类型的对象。



```
Get-Process -InputObject <Process[]> [-ComputerName <string[]>] [-Module] [-FileVersionInfo] [  
参数  
-ComputerName <string[]>  
是否必需? False  
位置? 已命名  
是否接受管道输入? True (ByPropertyName)  
参数集名称 InputObject, Name, Id  
别名 Cn  
动态? false  
-FileVersionInfo  
是否必需? False  
位置? 已命名  
是否接受管道输入? False  
参数集名称 Name, InputObject, Id  
别名 FV, FVI  
动态? false  
-Id <int[]>  
是否必需? True  
位置? 已命名  
是否接受管道输入? True (ByPropertyName)  
参数集名称 IdWithUserName, Id  
别名 PID  
动态? false  
-IncludeUserName  
-- More --
```

图9.15 验证-ComputerName参数支持的数据类型

回到之前起始部分，我们可以将执行结果通过管道传给Get-Member来展现命令A的输出结果。图9.16显示了这个结果。

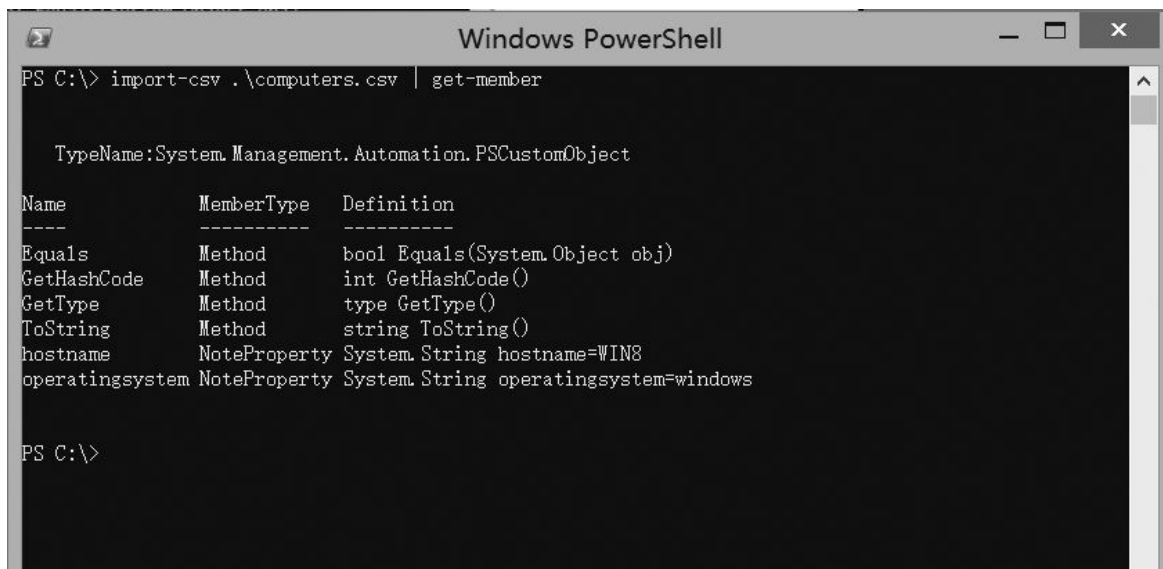


图9.16 Import-CSV命令产生PSCustomObject类型对象

Import-CSV的PSCustomObject类型输出并不是String，所以下的命令无法被执行。

```
PS C:\> Get-Process -ComputerName (Import-CSV .\Computers.CSV)
```

之后尝试从CSV文件中读取Host Name列，然后查看其输出结果，如图9.17所示。

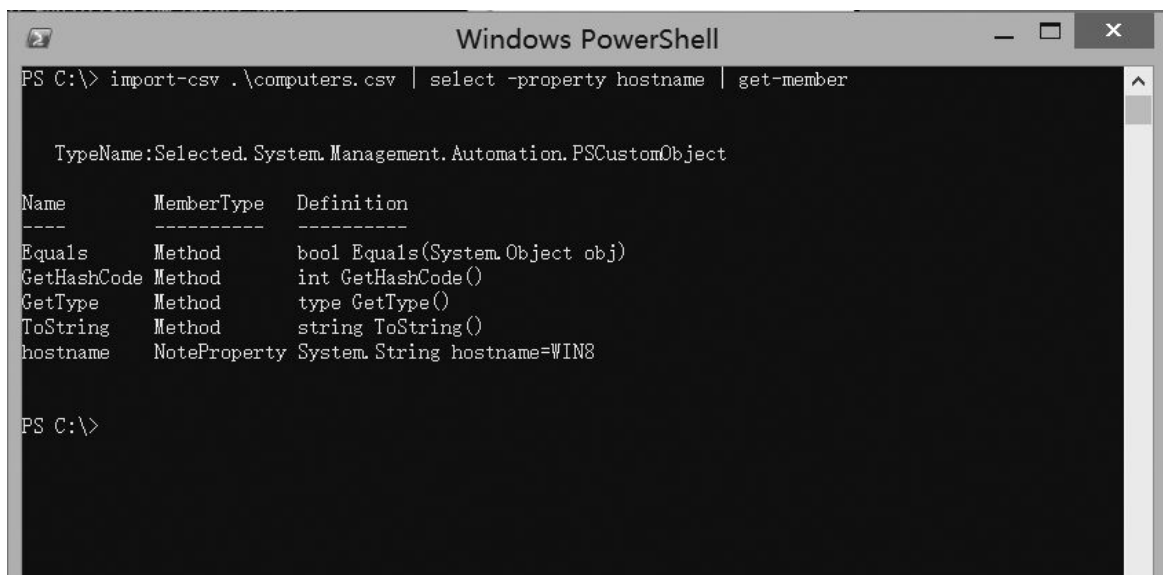


图9.17 选择单个属性，结果仍然是PSCustomObject类型

你得到了一个PSCustomObject类型对象。相比于之前的结果，它包含更少的属性。这也是Select-Object和-Property参数的一个特点。它并不会真正影响输出整个对象的行为。

但是-ComputerName这个参数的不会处理PSCustomObject对象，所以下面的这个命令仍然无法正常运行。

```
PS C:\> Get-Process -ComputerName (Import-CSV .\Computers.CSV |  
Select -Property HostName)
```

这也就使得-ExpandProperty参数有了用武之地。然后尝试加上该参数，并查看该命令执行的结果，如图9.18所示。

因为HostName属性中包含文本字符串，-ExpandProperty参数就可以将这部分值放入到一些简单的String对象中去，之后-ComputerName参数就可以处理这部分值了。翻译成脚本语言，如下所示：

```
PS C:\>Get-Process -ComputerName (Import-CSV .\Computers.CSV |  
Select -Expand HostName)
```

该技术功能非常强大。刚接触时，可能比较难以掌握，但是如果意识到一个属性是类似于盒子的概念，这将有助于我们掌握该技术。当使用Select-Property的时候，就会确定需要使用哪个盒子，但是也只是获取到盒子而已。当使用Select -ExpandProperty时，你就可以打开对应盒子，提取里面的内容，最后扔掉整个盒子，仅保留需要的内容。

```
Windows PowerShell
PS C:\> import-csv .\computers.csv | select -expand hostname | get-member

TypeName: System.String

Name      MemberType Definition
-----
Clone     Method      System.Object Clone(), System.Object ICloneable.Clone()
CompareTo Method      int CompareTo(System.Object value), int CompareTo(string strB
Contains  Method      bool Contains(string value)
CopyTo    Method      void CopyTo(int sourceIndex, char[] destination, int destinat
EndsWith  Method      bool EndsWith(string value), bool EndsWith(string value, Syst
Equals    Method      bool Equals(System.Object obj), bool Equals(string value), bo
GetEnumerator Method      System.CharEnumerator GetEnumerator(), System.Collections.IEn
GetHashCode Method      int GetHashCode()
GetType   Method      type GetType()
GetTypeCode Method      System.TypeCode GetTypeCode(), System.TypeCode IConvertible.G
IndexOf   Method      int IndexOf(char value), int IndexOf(char value, int startInd
IndexOfAny Method      int IndexOfAny(char[] anyOf), int IndexOfAny(char[] anyOf, in
Insert    Method      string Insert(int startIndex, string value)
IsNormalized Method      bool IsNormalized(), bool IsNormalized(System.Text.Normalizat
LastIndexOf Method      int LastIndexOf(char value), int LastIndexOf(char value, int
LastIndexOfAny Method      int LastIndexOfAny(char[] anyOf), int LastIndexOfAny(char[] a
Normalize Method      string Normalize(), string Normalize(System.Text.Normalizatio
PadLeft   Method      string PadLeft(int totalWidth), string PadLeft(int totalWidth
PadRight  Method      string PadRight(int totalWidth), string PadRight(int totalWid
Remove    Method      string Remove(int startIndex, int count), string Remove(int s
Replace   Method      string Replace(char oldChar, char newChar), string Replace(st
Split     Method      string[] Split(Params char[] separator), string[] Split(char[
StartsWith Method      bool StartsWith(string value), bool StartsWith(string value,
Substring Method      string Substring(int startIndex), string Substring(int startI
ToBoolean Method      bool IConvertible.ToBoolean(System.IFormatProvider provider)
ToByte    Method      byte IConvertible.ToByte(System.IFormatProvider provider)
```

图9.18 最终得到一个String类型的对象

9.8 动手实验

注意： 在本章实验环境，需要运行3.0版本的PowerShell或者之后版本的计算机。

再次提醒大家，在本章很短的时间内，我们讲解了很多重要的概念。巩固这些新学知识最好的办法就是立即使用它们。我们建议按照顺序依次完成下面的任务，因为这些任务逐层依赖，可以帮助我们复习学到的知识点，并且能帮助我们找到如何实践学到的这些知识。

为了让实验环节更有挑战性，我们强烈建议你测试Get-ADComputer命令。任何安装了Windows Server 2008 R2或者之后版本操作系统的域控制服务器都有默认安装该命令，但是实际上，在该环节，并不需要。你只需要了解到下面三点即可：

- `Get-ADComputer`命令包含一个`-Filter`参数；运行`Get-ADComputer-Filter*`会返回所在域中所有的计算机对象。
- 域中计算机对象都包含一个`Name`属性，该属性包含了计算机的名称信息。
- 域中计算机对象都会返回一个名为`ADComputer`的类型名称，也就是说，`Get-ADComputer`命令会返回`ADComputer`类型的对象。

这是你应该知道的三个知识点。请记住这几点，然后完成下面的任务。

注意： 我们并不会要求你真正去运行这些命令；相反，你需要判断这些命令是否可以正常执行，如果不能正常执行，请给出对应的原因说明。前面章节已经介绍了`Get-ADComputer`命令是如何工作的，然后该命令会返回何种类型的对象。你也可以通过帮助文件来查看其他命令可以处理的对象。

1. 下面的命令是否可以获取特定域中所有计算机上已经安装的Hotfix的清单？同时，请参照本章开头的格式，阐述其原因。

```
Get-HotFix -ComputerName (Get-ADComputer -Filter * |  
Select-Object -Expand Name)
```

2. 下面的命令是否可以从相同计算机上获取到HotFix列表呢？同时，请参照本章开头的格式，阐述其原因。

```
Get-ADComputer -Filter * |  
Get-HotFix
```

3. 下面第三个版本的命令是否可以获取到域中计算机上已经安装的HotFix清单？同时，请参照本章开头的格式，阐述其原因。

```
Get-ADComputer -Filter * |  
Select-Object @{l='ComputerName';e={$_.Name}} |  
Get-HotFix
```

4. 使用管道参数绑定来写一个命令获取域中每一台计算机上正在运行的进程的清单。不要使用括号。

5. 可以使用括号而不要使用管道输入方法来获取域中每一台计算机上已经安装的服务清单。

6. 微软有些时候可能忘记给一个Cmdlet添加管道参数绑定。例如，下面的命令是否可以获取域中每台计算机上的信息？请参照本章开头的格式，阐述其原因。

```
Get-ADComputer -Filter * |  
    Select-Object @{l='ComputerName';e={$_.Name}} |  
Get-WMIObject -Class Win32_BIOS
```

9.9 进一步学习

我们看到很多同学很难理解管道输入概念，主要是因为这个概念比较抽象。如果你觉得自己也是如此，那么请参考MoreLunches.Com网站。根据本书的封面或者名字去寻找，之后单击打开。找到“下载资源”部分，然后单独下载管道输入手册。将该手册打印多份，拿着一支笔，然后查看其中示例（比如Get-Service | Stop-Service）。该工作手册有逐步讲解整个管道输入流程的每一步。

第10章 格式化及如何正确使用

现在快速回顾一下：你已经知道PowerShell Cmdlets可以用于产生对象，并且这些对象通常含有比PowerShell默认显示更多的属性。你也已经知道如何使用“Gm”命令获取一个对象的所有属性，以及如何使用“Select-Object”去自定义你想看到的属性。到目前为止，你看到的基本上都是通过PowerShell的默认配置和规则把结果输出到显示器上（或者文件形式和硬拷贝格式）。本章将会介绍如何覆盖这些默认值并创建你自己的命令的输出格式。

10.1 格式化：让输出更加美观

因为PowerShell并不是完全成熟的用于生成管理报表的工具，所以读者不要因为前面的例子而产生误解。但是PowerShell的确能很好地收集计算机的信息，并以定制的格式输出结果。用于输出定制格式的方式称为格式化。

表面上看，PowerShell的格式化系统貌似很容易（大部分情况下也的确如此）。但是有时候一些需要技巧的方式会让你掉入陷阱中，所以希望你能明白它们是如何工作的，并且为什么要这样。本章不打算演示什么新命令，而是解释整个系统是如何工作的，并且你如何与它交互，另外还有需要注意的限制。

10.2 默认格式

现在执行一下我们熟悉的命令“Get-Process”，然后注意结果的列头部分。可以看到，它们并不是非常符合常规的属性名。取而代之的是一个固定的宽度、别名等。你是否意识到这些结果来自于某些配置文件？你可以在安装PowerShell的路径下找到其中一个名为“.format.pslxml”的文件。其中进程对象的格式化目录在“DotNetTypes.format.pslxml”中。

动手实验：接下来你需要一直打开PowerShell，以便跟随我们的脚步前进，并且从中理解格式化系统的底层结构。

下面我们先修改PowerShell的安装目录，并且打开“DotNetType.format.pslxml”文件。注意，别在这个文件中保存任何变更信

息。这个文件带有数字签名，即使一个简单的回车或者空格号，都会影响签名并阻止PowerShell从中获取信息。

```
PS C:\>cd $pshome
PS C:\>notepad dotnettypes.format.ps1xml
```

然后从中找出准确的类型并返回给“Get-Process”：

```
PS C:\>get-process | gm
```

接下来完成下面的步骤：

(1) 复制完整的类型名：**System.Diagnostics.Process**，并粘贴。可以用键盘光标高亮选中类型名，然后按回车键复制到粘贴板。

(2) 切换到记事本，然后按Ctrl+F组合键打开查找窗口。

(3) 在窗口中粘贴类型名，然后单击“查找下一个”。

(4) 你能找到的第一个对象一般是“ProcessModule”，这不是进程对象。所以继续查找下一个对象，直到找到“System.Diagnostics.Process”为止，如图10.1所示。

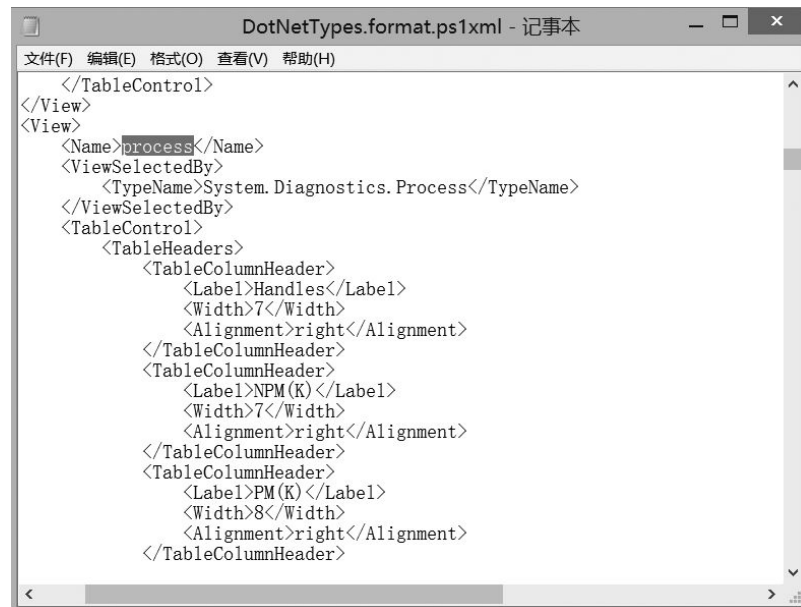


图10.1 在Windows记事本中定位进程视图

你现在看到的是在记事本中以默认形式显示一个进程的管理目录。稍微向下滚动一点，可以看到表视图的定义。这可能是你希望见到的结果，因为你已经知道如何把进程显示在一个多列的表中。从中可以看到熟悉的列名，如果再往下一点点，还能发现特定表中每列的展示属性，还有列宽及别名的定义等。浏览过后，关闭记事本，切记不要把信息保存到这个文件中，然后返回PowerShell。

当运行“Get-Process”，在Shell中会发生下面的事情：

- (1) Cmdlet把类型为“System.Diagnostics.Process”的对象放入管道。
- (2) 在管道的末端是一个名为“Out-Default”的隐藏的Cmdlet。这个Cmdlet的作用是把需要运行的命令全部放入管道中，注意此Cmdlet总会存在。
- (3) “Out-default”把对象传输到“Out-Host”，原因是PowerShell控制台默认把输出结果显示到机器所在的显示屏上（称为host）。理论上，可以写一个Shell把文件或打印机作为默认输出设备，但是目前为止还没听说过有人这样做。
- (4) 大部分“Out-Cmdlets”不适合用在普通对象中，而主要用于特定格式化指令上。所以当“Out-Host”看到那些普通对象时，会把它们传递给格式化系统。
- (5) 格式化系统以其内部格式化的规则检查对象的类型（我们将在下面介绍）。然后用这些规则产生格式化指令，最终传输回“Out-Host”。
- (6) 一旦“Out-Host”发现已经生成了格式化指令，就会根据这个指令产生显示到屏幕上的结果。

上面提到的内容也会在你手动指定“Out-Cmdlet”的时候发生。比如运行“Get-Process | Out-File procs.txt”和“Out-File”时，PowerShell会看到你发送了一些普通对象。它会把这些对象发给格式化系统，然后创建格式化指令后回传给“Out-File”。“Out-File”基于这些指令创建格式化后的文本文件。所以在需要把对象转换成用户可读的文本输出格式时，格式化系统就会起到作用。

在上面12步中，PowerShell依赖于什么格式化规则？其中第一个规则是系统会检查对象类型是否已经被预定义视图处理过。也就是我们见到

的“DotNetType.format.ps1xml”：针对进程的对象。PowerShell中预装了其他的“.format.ps1xml”文件，在Shell启动时自动加载。你也可以创建自己的预定义视图，但是这部分超出了本书范围。

格式化系统对特定对象的类型查找相应的预定义视图，在本例中也就是查找处理“System.Diagnostics.Process”对象的视图。

如果没找到对应的视图会发生什么？比如运行：

```
Get-WmiObject Win32_OperatingSystem | Gm
```

选中对象类型的名字（最少选择“Win32_OperatingSystem”部分），然后尝试在其中一个“.format.ps1xml”文件中查找它。为了节省时间，我们直接告诉你是找不到的。

这是格式化系统下一步要做的事情，我们也可以称之为第二个格式化规则：格式化系统寻找是否有针对这个对象类型的“default display property set”。这些可以在另外一个配置文件“Types.ps1xml”中找到。现在继续用记事本打开（记住别保存任何修改），然后使用查找功能定位“Win32_OperatingSystem”。一旦找到它之后，向下滚动一点点，就可以看到“DefaultDisplayPropertySet”，如图10.2所示，注意下面列出的6个属性。



图10.2 在记事本中定位DefaultDisplayPropertySet

现在返回PowerShell，然后运行：

```
Get-WmiObject Win32_OperatingSystem
```

结果是不是看起来很熟悉？这些属性单独看起来的确如此，因为它们来自于默认的“Types.ps1xml”文件。如果格式化系统找到一个“default display property set”，会把这个属性集用于下一步的决策。如果没有找到，那么下一步的决策将考虑所有对象的属性值。

接下来是决策，即格式化第三个规则——用于决定输出的样式。如果格式化系统将显示4个或以下的属性，将决定以表格形式展现。如果有5个或以上的属性，会使用列表形式。这就是“Win32_OperatingSystem”对象的结果不以表格显示的原因。它的结果有6列，所以以列表形式展示。其中原理就是当属性超过4个时，可能不能很好地展示到一个未经处理的实时表格中。

现在你已经了解格式化是如何工作的，并且明白了大部分“Out-Cmdlets”会自动触发格式化系统，以便能按需找到格式化指令。下面看看我们如何控制格式化系统，并且覆盖默认值。

10.3 格式化表格

在PowerShell中，有4种格式化的Cmdlets。我们将介绍日常使用最多的3种（第4种会在本节结尾的“补充说明”中简要介绍）。首先是“Format-Table”，其别名为“Ft”。

如果你查看“Format-Table”的帮助文档，可以发现这个命令有很多参数。我们将演示其中最常用的几个。

- **-autoSize**——通常情况下，PowerShell会根据你的屏幕生成和填充表格（除非存在一个预定义视图，如针对进程的）。这意味着结果集会只有少量的列，并且列与列之间的空隙较大，并不总是引人注目。通过使用“-autoSize”，可以强制结果集仅保存足够的列空间，使得表格更加紧凑，但是会耗费少量的时间让Shell产生输出，因为对于每个对象都需要在格式化的过程中搜寻每个列的最大值。尝试在下面的语句中对比是否有“-autoSize”参数的结果：

```
Get-WmiObject Win32_BIOS | Format-Table -autoSize
```

- **-property**——该参数接收一个以逗号分割的希望包含在结果表格中的属性列表。这些属性是不区分大小写的，但是Shell会使用你的输出作为列头。如果你希望输出格式以期望的形式展现，需要规定好名字的格式（如使用“CPU”替代“cpu”）。另外，这个参数也接受通配符，可以使用“*”替代的所有属性，或者使用如“c*”标识所有以c开头的属性名。但是需要注意的是，这个Shell依旧使用仅能填充到一个表格的属性作为展示，并不是你指定的都会输出。这个参数是依赖位置的，所以可以不输入参数名，只需要在第一个位置提供属性列表即可。尝试运行下面的语句（结果见图10.3）：

```
Get-Process | Format-Table -property *  
Get-Process | Format-Table -property ID,Name,Responding -autoSize  
Get-Process | Format-Table * -autoSize
```

- **-groupBy**——每当指定的属性值变更时，创建一个具有新列头的结果集。这个参数只在你第一次对具有相同属性的对象进行排序时工作良好。可以通过例子去理解参数是如何工作的：

```
Get-Service | Sort-Object Status | Format-Table -groupBy Status
```

- **-wrap**——如果Shell需要把列的信息截断，会在列尾带上 (...) 以便标识信息被截断。这个参数能使Shell把信息收起，会使你的表变长，但是当你需要查看的时候却很有用。下面是例子：

```
Get-Service | Format-Table Name,Status,DisplayName -autoSize -wrap
```

```

PS C:\> ps | ft -auto

```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
161	20	3344	4548	100		2268	AppleMobileDeviceService
85	6	1396	1384	114		5552	appverif
85	6	1396	1388	114		5568	appverif
85	6	1392	1388	114		5584	appverif
84	9	9072	2708	65		5608	appverif
72	6	972	2020	17		1844	AsLdrSrv
155	15	3356	9744	99	0.06	7532	AsusTPCenter
46	6	1156	4448	50	0.02	6676	AsusTPHelper
136	11	1976	8604	82	0.08	9008	AsusTPLoader
65	5	804	880	42		5800	AtBroker
65	5	792	880	42		5816	AtBroker
65	5	808	884	42		5840	AtBroker
65	5	808	892	42		5872	AtBroker
100	13	2336	10364	114	0.14	3640	ATKOSD2
140	9	6124	9080	39	0.36	7148	audiodg
122	10	15628	14964	98		3652	bootim
122	10	15844	15076	98		5216	bootim
122	10	15632	15124	98		5576	bootim
122	10	15716	15100	98		5748	bootim
116	12	4568	4600	78		5208	calc
116	12	4584	4632	78		5624	calc
116	12	4580	4612	78		5704	calc
116	12	4584	4604	78		5808	calc
70	8	1424	5944	74	0.03	10296	caller64
90	7	1248	1508	30		5144	CameraSettingsUIHost
92	7	1260	1508	30		5212	CameraSettingsUIHost

图10.3 创建关于进程的自动大小表格

动手实验：你应该运行上面提到的所有Shell，然后尝试通过混合这些技术，体会它们是如何运作和如何排序的。

10.4 格式化列表

有时候你需要水平地把数据展现到一个表中，此时使用列表就很有用。“Format-List”是时候用上了，注意你可以使用其别名：Fl。

这个Cmdlet也支持一些类似的参数，如“Format-Table”，包括“-property”。实际上，Fl是另外一个展示对象属性的方法，和Gm不一样。Fl也同样显示这些属性的值，以便你可以看到每个属性包含的信息：

```
Get-Service | Fl *
```

图10.4展示了命令的结果。我们经常使用Fl作为发现对象属性的候选方案。

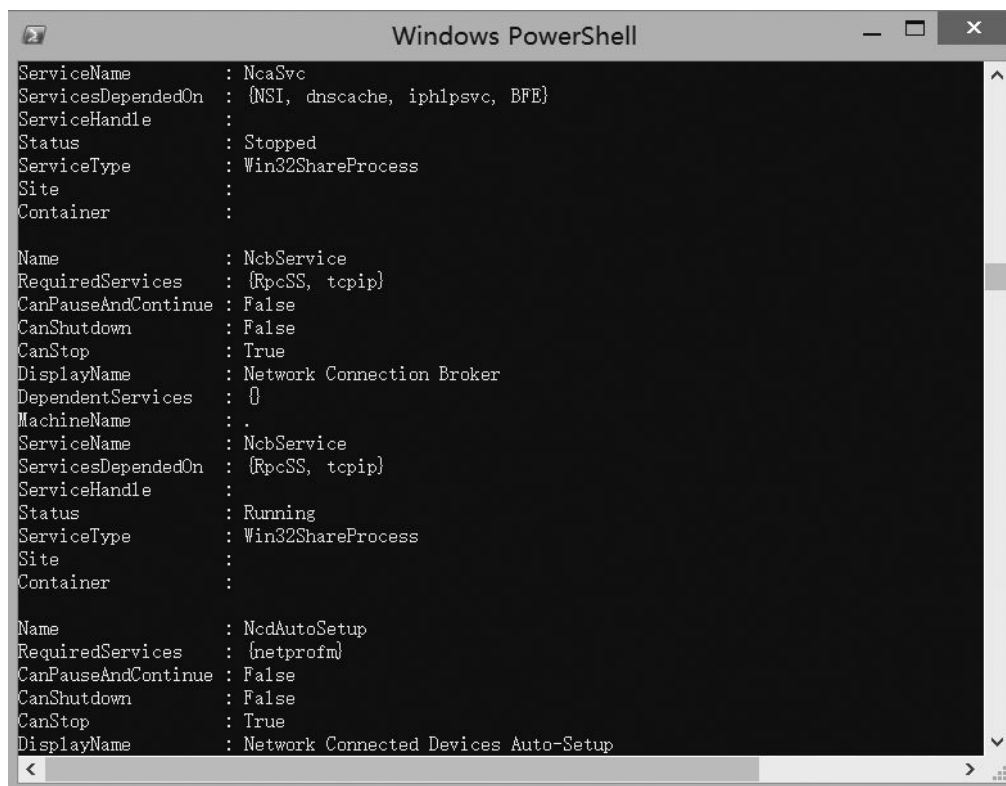


图10.4 检查显示在列表中的服务

动手实验： 查阅“Format-List”的帮助文档，并尝试体会它们参数的异同。

10.5 宽度的格式化

我们将展示的最后一个是“Format-Wide”（或者别名Fw），用于展示一个宽列表。它仅展示一个单独属性的值，所以它的“-property”参数仅接受一个属性名，而不是列表，并且不接受通配符。

默认情况下，“Format-Wide”会查找对象的“Name”属性，因为“Name”是广泛使用的属性并且通常包含有用信息。默认显示只有两列，但是“-columns”参数可以用于指定更多的列：

```
Get-Process | Format-Wide name -col 4
```

图10.5展示了其结果。


```
Get-Process |  
Format-Table Name,  
@{n='VM(MB)';e={$_.VM / 1MB -as [int]}} -autosize
```

图10.6展示了前面命令的结果。其实我们做了一点小动作，用了一些之前没提到过的技术。下面我们稍微说明一下。

- 我们从“Get-Process”开始，相信你已经很熟悉这个命令。如果你运行“Get-Process | Fl *”，你会看到“VM”的属性是以字节为单位，虽然默认的表格视图显示并不如此。
- 我们从进程的“Name”属性开始讨论“Format-Table”。

Name	VM (MB)
AppleMobileDeviceService	100
appverif	114
appverif	114
appverif	114
appverif	65
AsLdrSrv	17
AsusTPCenter	99
AsusTPHelper	50
AsusTPLoader	82
AtBroker	42
AtBroker	42
AtBroker	42
AtBroker	42
ATKOSD2	114
audiodg	39
bootim	98
bootim	98
bootim	98
bootim	98
calc	78
calc	78
calc	78
calc	78
caller64	74
CameraSettingsUIHost	30
CameraSettingsUIHost	30
CameraSettingsUIHost	30
CameraSettingsUIHost	30

图10.6 创建一个定制的结果，统计表列MB值

- 接着，我们创建一个以“VM(MB)”为标签的列，其值或者表达式是针对对象的常规VM属性并且除以1 MB。在PowerShell中的斜线是除法操作。另外，“KB”“MB”“GB”“TB”和“PB”缩写分别代表kilobyte、megabyte、gigabyte、terabyte和petabyte。
- 除法运算的结果会自带小数点组件，“-as”操作符可以帮助我们帮数据结果从浮点型转换成整型（如指定[int]）。这个Shell会适当地向上或者向下取整，以便显示适当的结果。其最终结果是没有小数的数值。

补充说明

建议你重复执行下面的例子：

```
Get-Process |  
Format-Table Name,  
@{n='VM(MB)';e={$_.VM / 1MB -as [int]}} -autosize
```

不过这次不要在一行中全部输入，而是按照上面的格式输入。你会发现，当你输入完第一行之后（也就是以管道符结尾），PowerShell会出现一个提示符。因为你以管道符结尾，所以Shell知道你准备输入更多的命令，直到你以花括弧、引号和括号结尾为止。

如果你不想以这种“扩展输入模式”输入，按Ctrl+C组合键来忽略。在这种情况下，输入第二行文本并按回车键，然后继续输入第三行，再按回车键。你必须在这种模式下在一个空行中按最少一次回车键，以便告知Shell你已经完成输入。然后Shell会逐行顺序执行你的输入。

在这里提到除法运算及其数据类型修改的技巧，是因为在输出美观的结果时非常有用。我们不打算在本书中花费过多时间在这些操作符中（仅仅介绍“*”代表乘法，“+/-”分别代表加减运算）。

和“Select-Object”不一样，它的哈希表仅接受一个名字和表达式键（对于名字，可以为N、L和Label；对于表达式，可以接受E）。为了可视化展示，“Format-”命令可以接受额外的关键字。这些关键字对于“Format-Table”十分有效。

- **FormatString**：指定一个格式化代码，让结果根据这个代码格式化，主要用于数值型和日期型数据。可以到MSDN的“Formatting Types”（<http://msdn.microsoft.com/en-us/library/26etazsy.aspx>）页中查看标准数值型和日期型格式的可用代码。另外，这里还包含了自定义的格式。
- **Width**：指定列宽。
- **Alignment**：指定列的对齐格式，可以为左对齐或者右对齐。

使用这些关键字修改上面的代码，能使结果更加易读和美观。

```
Get-Process |  
Format-Table Name,  
@{n='VM(MB)';e={$_.VM};formatstring='F2';align='right'} -autosize
```

现在我们并不需要使用除法，因为PowerShell会以两个小数位并右对齐的形式格式化结果。

10.7 输出到文件、打印机或者主机上

一旦对象被格式化，你必须决定结果的去向。

如果命令以“Format-Cmdlet”结束，格式化指令将按“Format-Cmdlet”的“Out-Default”创建，也就是以“Out-Host”显示结果到显示屏。

```
Get-Service | Format-Wide
```

你可以手动在上面命令中输入“Out-Host”，并以管道符连接。实际上运行了下面语句：

```
Get-Service | Format-Wide | Out-Host
```

另外一种方式是用管道把格式化指令的“Out-File”或“Out-Printer”定位到文件或者打印机中，比如从“常见的困惑”中看到，只有这三个“Out-”Cmdlet才可以跟在“Format-”Cmdlet后面。

请记住，“Out-Printer”和“Out-File”都有默认的输出宽度，意味着它们的结果宽度看上去可能和显示器上看到的不一致。这些Cmdlets允许你使用“-width”参数控制宽度，输出你想要的表格。

10.8 另外一个输出：网格

“Out-GridView”提供了另一种有用的输出功能。但是注意，这并不是技术上的格式化。实际上，“Out-GridView”完全绕过了格式化子系统。它不需要调用“Format-”Cmdlets，不产生格式化指令，没有文本结果输出到控制台窗口。“Out-GridView”不接收“Format-”Cmdlet的输出，仅接收其他Cmdlets输出的对象。

图10.7显示了网格的样子。

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
161	20	3344	4548	100		2,268	AppleMobileDeviceService
85	6	1396	1384	114		5,552	appverif
85	6	1396	1388	114		5,568	appverif
85	6	1392	1388	114		5,584	appverif
84	9	9072	2708	65		5,608	appverif
72	6	972	2020	17		1,844	AsLdrSrv
155	15	3356	9744	99	0.06	7,532	AsusTPCenter
46	6	1156	4448	50	0.02	6,676	AsusTPHelper
136	11	1976	8604	82	0.08	9,008	AsusTPLoader
65	5	804	880	42		5,800	AtBroker
65	5	792	880	42		5,816	AtBroker
65	5	808	884	42		5,840	AtBroker
65	5	808	892	42		5,872	AtBroker
100	13	2336	10364	114	0.14	3,640	ATKOSD2
140	9	6128	9080	39	0.44	7,148	audiodg
122	10	15628	14964	98		3,652	bootim
122	10	15844	15076	98		5,216	bootim

图10.7 “Out-GridView” Cmdlets的结果

10.9 常见误区

正如本章开头所说，PowerShell的格式化系统对新手来说存在不少陷阱。根据过往经验，我们整理出两个主要的注意事项，希望能帮助读者更好地避开这些陷阱。

10.9.1 总是以右对齐来格式化

切记：format right。你的“Format-”Cmdlet应该是“Out-File”或者“Out-Printer”作为仅有表达式时的命令行的最后一个命令。其原因是“Format-”Cmdlets产生格式化指令，仅有“Out-”Cmdlet能合理地处理这些指令。如果一个“Format-”Cmdlet作为命令行的结尾，指令将使用“Out-Default”（总为管道的结尾）即指向“Out-Host”，这会导致非预期的格式化。

为了演示，执行以下命令：

```
Get-Service | Format-Table | Gm
```

你会看到如图10.8所示，“Gm”没有显示你希望的服务对象的信息，因为“Format-Table”Cmdlet并不输出服务对象。它处理掉你通过管道传输的服务对象，并且输出格式化指令——这正如你看到的“Gm”显示的结果。

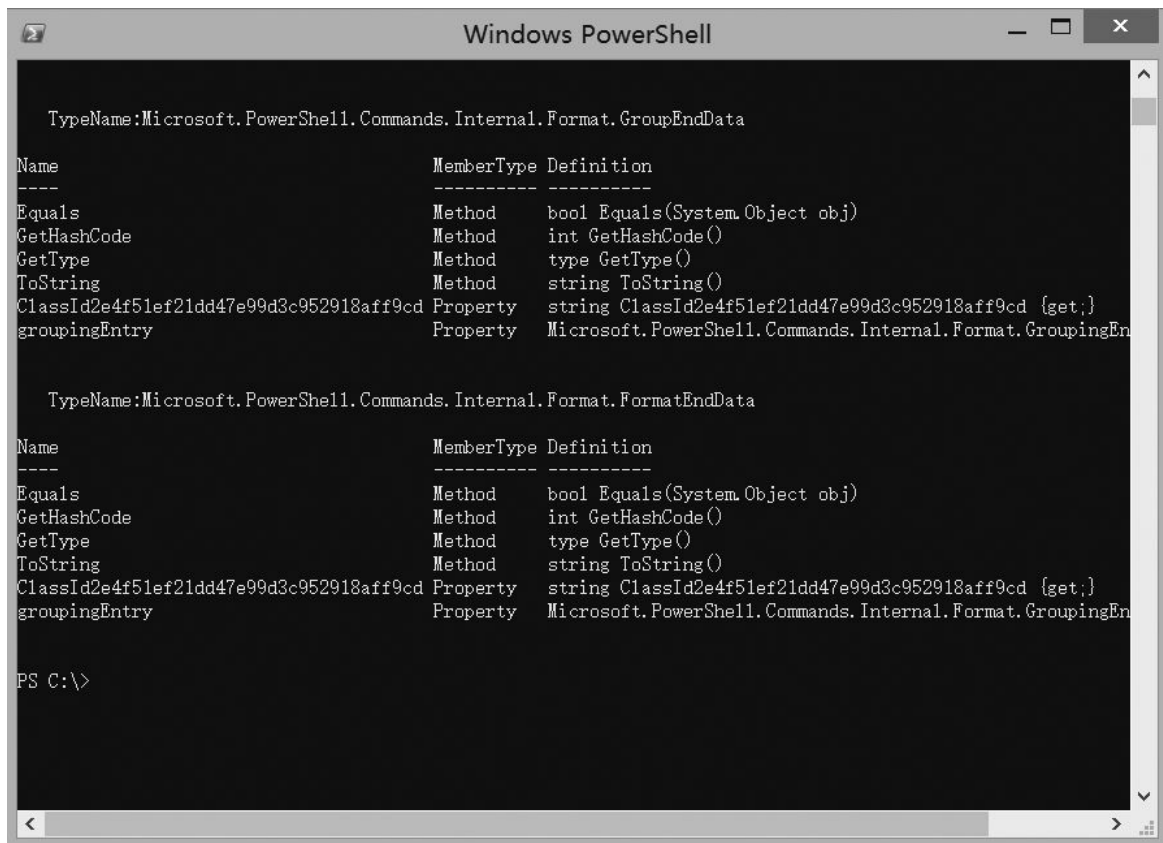


图10.8 格式化Cmdlets产生的特定格式化指令，可见其易读性不高

动手实验:

```
Get-Service | Select Name,DisplayName,Status | Format-Table |
ConvertTo-HTML | Out-File services.html
```

接着用IE打开Services.html文件，你可以看到让你抓狂的结果。你并没有把服务对象用管道传输到“ConvertTo-HTML”中，你只是传输了格式化指令，然后转换成HTML。这里演示了为什么一旦使用了某个“Format-”Cmdlet，要么把它作为命令行的最后一个Cmdlet，要么必须出现在“Out-File”或者“Out-Printer”的前面。

同样，我们知道“Out-GridView”也是不寻常的（最起码针对“Out-”Cmdlet来说），因为它不接受格式化指令且仅接受常规对象。可以用下面命令查看差异：

```
PS C:\>Get-Process | Out-GridView
```

```
PS C:\>Get-Process | Format-Table | Out-GridView
```

这就是为什么我们额外提醒“Out-File”和“Out-Printer”是仅有的需要跟在“Format-”Cmdlet后面的命令（技术上，“Out-Host”也可以跟在“Format-”Cmdlet后面，但是没有必要，因为以“Format-”Cmdlet结尾的命令无论如何都会输出到“Out-Host”上）。

10.9.2 一次一个对象

另外一件需要避免的事就是把多种对象放入管道。格式化系统先在管道中查找第一个对象，然后使用定义的格式处理这个对象。如果管道包含两个或以上的对象，那么结果可能不是你想要的。

比如运行：

```
Get-Process; Get-Service
```

其中分号允许我们把两个命令合并在一个命令行中，而不是把第一个命令的输出以管道形式传入第二个命令。这意味着两个命令是单独运行的，但是会把它们的输出传到相同的管道中。如果你动手运行或者查看图 10.9，会看到第一个命令的输出是合理的，但是当显示服务对象时，输出结果会变成另一个格式，而不是使用相同的表格，此时PowerShell会使用列表显示。PowerShell的格式化系统并不用于把多个对象和结果按你期望的形式合并。

那么如何把两个结果集以单一格式显示呢？此时可以使用本书没有提到的一些高级话题来处理这个需求，从而让格式化系统能合理地美化结果。

注意:

本实验需要PowerShell v3或以上版本。

尝试独立完成下面任务:

1. 显示一个表格, 包含进程名、ID, 不管这些进程是否对Windows响应 (“**Responding**”属性中能找到这些信息)。尽可能使这些信息横向填满整个窗口, 但不要使任何信息截断。

2. 显示一个表格, 包含进程名、ID。表中的列还要包含虚拟内存和物理内存的使用情况, 以MB为标识单位。

3. 使用“**Get-EventLog**”显示所有可用事件日志的列表 (提示: 你需要查看帮助文档, 以便找到能完成这个任务的信息)。并把这些信息格式化成表, 日志需要显示名字和保留期限, 分别以“**LogName**”和“**RetDays**”表示。

4. 显示一个关于服务的列表, 针对服务的正在运行和结束分别显示。而正在运行的服务需要优先显示 (提示: 你可能需要使用“**-groupBy**”参数)。

10.11 进一步学习

这是针对格式化系统实验的好时机。尝试使用三个主要的“**Format-***” Cmdlets创建不同格式的输出。在下一章将频繁要求你使用特定的格式化形式, 所以你需要在这一章中锻炼相关技能, 并且记好本章中的常用参数。

第11章 过滤和对比

到目前为止，我们使用Shell向你展示了不同类型的输出：所有进程、所有服务、所有事件日志条数、所有补丁。但是这些类型的输出并不总是你想要的结果。通常你会想要缩小结果到你感兴趣的几项。你将在本章学会该知识。

11.1 只获取必要的内容

Shell提供了两种方式缩小结果集，它们都归结为过滤。第一种方式：尝试指定Cmdlet命令只检索指定的内容。第二种方式：采用迭代的方法，通过第一个Cmdlet获得所有结果，并使用第二个Cmdlet过滤掉不想要的东西。

按道理，应该使用第一种方式：我们称之为尽可能提前过滤。这让Cmdlet更加容易知道你想要的。例如，使用Get-Service，你可以告诉它你想要的服务名：

```
Get-Service -name e*, *s*
```

如果你想让Get-Service只返回正在运行的服务，而不考虑它们的服务名称，该Cmdlet就无法做到这一点，因为它没有提供相关的参数来指定该信息。

同理，如果你使用微软的活动目录模块，所有的Get- Cmdlets命令都提供了-filter参数。通过-filter*，你可以获取所有的对象。我们不建议这样使用，因为加载它将增加域控制器压力。可以指定类似下面清晰的条件说明：

```
Get-ADComputer -filter "Name -like '*DC'"
```


再者，上述技巧的优势在于该Cmdlet只检索匹配的对象。我们称之为左过滤技术。

11.2 左过滤

“左过滤”意味着尽可能把过滤条件放置在左侧或靠近命令行的开始部分。越早过滤不需要的对象，就越能减轻其他Cmdlets命令的工作，并且能减少不必要的信息通过网络传输到你的电脑。

左过滤技术的缺点是每个Cmdlet都可以通过自己的方式指定过滤，并且每个Cmdlet都会有不同的过滤方式。例如Get-Service，你只能通过Name属性过滤服务。但是使用Get-ADComputer，你可以根据computer对象可能存在的任何活动目录属性进行过滤。在有效使用左过滤技术之前，你需要学习不同Cmdlet的各种操作。这可能意味着学习的道路有些崎岖，但是你却会得到更好的性能。

当无法通过一个Cmdlet就可以完成你所需的所有过滤时，你可以使用一个叫作Where-Object（它的别名为Where）的核心PowerShell Cmdlet命令。这是一个通用的语法。当需要检索的时候，使用它过滤任何类型的对象，并把它放入管道。

为了使用Where-Object，需要学会告诉Shell如何过滤出你想要的信息，这还包括使用Shell的对比操作符。有趣的是，一些左过滤技术中使用了相同的对比操作符，如活动目录模块下Get-Cmdlet命令的-filter参数，这就是一箭双雕。但是有些Cmdlet命令（如Get-WmiObject，我们将在后面的章节中讨论）使用了完全不同的过滤和对比方式，当我们讨论这些Cmdlet命令的时候再做介绍。

11.3 对比操作符

在计算机中，对比总是涉及两个对象或者两个值来并测试它们彼此之间的关系。可能是测试它们是否相等或者是否其中一个比另外一个大，或者它们是否匹配某个文本表达式。这就需要使用对比操作符来完成对关系的测试。测试的结果总是返回一个布尔值：true或false。换句话说，测试结果或者满足你指定的条件，或者不满足。

PowerShell使用如下对比操作符。请注意，当对比文本字符串时会忽略大小写。这意味着大写字母和小写字母是等价的。

- **-eq**——相等，例如`5 -eq 5`（返回true）或者`"hello" -eq "help"`（返回false）。
- **-ne**——不等于，例如`10 -ne 5`（返回true）或者`"help" -ne "help"`（返回false，因为它们实际上相等的，这里测试它们是否不相等）。
- **-ge**和**-le**——大于或等于，小于或等于，例如`10 -ge 5`（返回true）或者`Get-Date -le '2012-12-02'`（这取决于你运行该命令的时间，这展示了日期也是可以比较的）。
- **-gt**和**-lt**——大于和小于，例如`10 -lt 10`（返回false）或者`100 -gt 10`（返回true）。

对于字符串的对比，如果需要区分大小写，可以使用下面的集合：**-ceq**，**-cne**，**-cgt**，**-clt**，**-cge**，**-cle**。

如果想一次比较多个对象，可以使用布尔运算符**-and**和**-or**。通常在每个子表达式两边加上圆括号，使得表达式更容易阅读。

- `(5 -gt 10) -and (10 -gt 100)` 返回false，因为一个或两个子表达式返回值为false。
- `(5 -gt 10) -or (10 -lt 100)` 返回true，因为最后一个子表达式返回值为true。

另外，布尔值**-not**对true和false取反。在处理一个变量或者已经包含true或false的属性时，这可能会有用。而你想测试相反的条件。例如，false如需要测试一个进程是否没有响应，可以这样做（使用\$__作为进程对象的容器）：

```
$_.Responding -eq $False
```

Windows PowerShell定义了\$False和\$True来表示false和true的布尔值。另外一种书写方式如下：

```
-not $_.Responding
```

因为Responding通常包含true和false，该-not让false取反变为true。如果进程没有响应，意味着Responding返回false。然而上面的比较却返回true，这就暗示着该进程“没有响应”。我们更喜欢使用第二种方式，因为在英语的阅读习惯中，它更接近我们的测试内容：“我想看看这个进程是否没有响应”。有些时候，你可以看到-not运算符简写为感叹号（!）。

当你需要对比文本字符串时，还有其他几个有用的对比运算符：

- -like接受*作为一个通配符，所以可以对比："Hello" -like "*ll*"（返回true）。相反，运算符为-notlike。它们都是忽略大小写的。区分大小写可以使用-clike和-cnotlike。
- -match用于文本字符串与正则表达式进行比较。-notmatch是个逻辑上的反义词。并且正如你所想，-cmatch和-cnotmatch提供了区分大小写语法。正则表达式超出了本书的讨论范围。

Shell的好处是你可以在命令行运行上面几乎所有的测试（除了前面提到的\$_占位符，它不能独立运行，但是你可以在下一节中看到它是如何运行的）。

动手实验：继续尝试上述比较操作符示例的部分或全部，在一行中输入5 -eq 5并敲回车键，看看返回的内容。

在about_comparison_operators的帮助文件中可以找到其他可用的对比运算符，你将在本书的第25章中了解其他部分运算符。

补充说明 如果Cmdlet命令不使用11.3节中讨论的PowerShell形式的比较运算符，可以使用高中或大学（甚至是工作中）学过的更加传统的编程语言形式的比较运算符。

- = 等于
- <> 不等于
- <= 小于或等于
- \>= 大于或等于
- \> 大于
- < 小于

如果支持布尔运算符，通常关键字是AND和OR。有些Cmdlet命令可能提供类似LIKE的运算符。例如，通过-filter参数可以找到Get-WmiObject支持的所有运算符。当我们在第14章讨论该Cmdlet时，会重现这个列表。

每个Cmdlet的设计者挑选如何（以及是否需要）处理过滤，通过查看该Cmdlet的完整的帮助通常可以获得能完成设计者期望的示例，包括帮助文件末尾附近的用法示例。

11.4 过滤对象的管道

当已经写好一个比较表达式，可以在哪里使用它？使用比较只是我们的概括语言。它可以与一些Cmdlet的-filter参数共同使用，也许是最引人注目的活动目录中模块的GET-的Cmdlet。它也可以与Shell的通用过滤Cmdlet命令Where-Object一起使用。

例如，你是否想过滤掉其他信息，只留下正在运行的服务？

```
Get-Service | Where-Object -filter { $_.Status -eq 'Running' }
```

-filter参数是一个位置参数，这意味着你经常看到很多命令没有指定这个参数，而它的别名为Where。

```
Get-Service | Where { $_.Status -eq 'Running' }
```

如果你习惯大声阅读上面代码，这会听起来合情合理：“where status equals running”。这就阐述了它的工作原理：当你传递多个对象到Where-Object时，它会检查每个对象从而进行过滤。一次只放置一个对象到占位符\$_，接着运行对比来查看返回的是true还是false。如果是false，该对象就会被移除管道。如果对比返回true，该对象就会从Where-Object传输到下一个Cmdlet的管道中。在上面的例子中，下一个Cmdlet命令是Out-Default，通常这是最后一个管道（在第8章已经讨论过），接着开始使用格式化进程显示输出。

占位符\$_是个特殊的产物：之前已经见过（在第10章），你将在一个或更多的上下文看到它。该占位符只能在PowerShell能查找的特定位置中使用。在我们的示例中，该占位符恰好是在其中一个特定位置。正如你在第10章学习到的，句号用于告诉Shell不是比较整个对象，而是只比较对象的Status属性。

希望你开始看到Gm派上用场了。它可以让你快速并且以方便的方式发现一个对象中包含的所有属性，这样你就可以马上使用这些属性进行类似上面的比较。始终牢记，PowerShell输出的列标题不总是跟属性名一模一样的。例如，运行Get-Process，可以看见一个叫作PM(MB)的列；运行Get-Process | Gm，发现实际的列名是PM。这是一个重要的区别：总是使用Gm验证属性名称，不要使用Format-这个Cmdlet命令。

补充说明

PowerShell v3为Where-Object引入了一个新的“简写”语法。当只有一个比较的时候可以使用该语法。如果需要对比多个子项，依旧得使用本小节中提到的原始语法。许多人争论这个简写语法是否有所帮助。类似下面的内容：

```
Get-Service | Where Status -eq 'Running'
```

显然，该写法更容易阅读：免除了花括号{}并且不需要使用看起来尴尬的占位符\$_。但是这个新语法不是意味着你可以忽略掉旧的语法，因为在更复杂的比较中需要使用它。

```
Get-WmiObject -Class Win32_Service |  
Where { $_.State -ne 'Running' -and $_.StartMode -eq 'Auto' }
```

而且，在互联网上6年的时间内，所有有价值的例子都是使用旧语法的。这意味着你需要知道怎么使用它们。你也必须知道该新语法，因为它现在会开始出现在开发人员的例子中。不必知道两套语法是不是已经足够“简化”，但至少你知道什么是什。

顺便说一下，我们承认上述命令并不是一个最好的例子，因为可以使用Get-WmiObject的-Filter参数，这样会更加高效。但是我们使用这个的目的是想说明和指出Where-Object的“旧”语法依然有用武之地。

11.5 迭代的命令行模式

我们现在想为你简单介绍PowerShell迭代命令行模型或者称为PSICLM（没有理由为它创建一个首写字母的缩写，但是它的读音却很有趣）。PSICLM的核心思想在于你不需要一开始就创建一个大而复杂的命令行，而是从简单的开始。

比方说，你想计算正在使用虚拟内存的十大进程占用的虚拟内存总量。如果这些进程中包含了PowerShell进程，而又不想在结果中包含该进程，快速罗列出几个需要的步骤：

- (1) 获取进程列表；
- (2) 排除PowerShell进程；
- (3) 按照虚拟内存进行排序；
- (4) 只保存前10个或者最后10个，这取决于我们的排序方式；
- (5) 把剩下进程的虚拟内存相加。

我们相信你知道如何完成前3个步骤，第4个步骤完全可以使用我们的老朋友：Select-Object。

动手实验：花几分钟时间阅读Select-Object的帮助文档。你是否能找到让你在一个集合中保留第一个或最后一个对象的任何参数？

希望你能找到答案。

最终，需要把所有虚拟内存相加。这是需要寻找新Cmdlet的地方，或许可以通过Get-Command或Help加上通配符来寻找。可以尝试add关键字，或者sum关键字，甚至是Measure关键字。

动手实验： 看看你能不能找到一个可以计算类似虚拟内存总量的命令。使用**Help**或**Get-Command**加上*通配符。

当你尝试这些小任务（而不是提前阅读答案），这会让自己变成一个**PowerShell**专家。一旦你觉得自己有答案了，你可能开始使用迭代的方法。

一开始，你需要获取所有的进程，这很容易满足：

```
Get-Process
```

动手实验： 跟随该Shell，并运行这些命令。验证每一个输出，看看你是否能预测下一次迭代的命令你需要改变什么。

下一步，过滤掉不需要的进程。记住，“左过滤”意味着你想尽可能在靠近开始命令行的地方进行过滤。在该示例中，将使用**Where-Object**来进行过滤，因为我们希望它成为下一个**Cmdlet**。虽然效果没有在第一个**Cmdlet**命令就进行过滤的好，但是总好过在最后的管道中才过滤。

在该Shell中，按键盘上的向上箭头键找回你最后的命令，并输入下面的命令。

```
Get-Process | Where-Object -filter { $_.Name -notlike  
'powerShell*' }
```

我们不确定进程名是“**powerShell**”或“**powerShell.exe**”，所以使用通配符来包含这两种可能。进程名称不像上面的所有进程将会留在管道中。

运行并测试，接着继续使用键盘上的向上箭头键找回上次命令并加上后面的部分。

```
Get-Process | Where-Object -filter { $_.Name -notlike  
'powerShell*' } |
```

```
Sort VM -descending
```

敲回车键可以验证你的输入，而键盘上的向上箭头键可以和后面的命令进行拼接。

```
Get-Process | Where-Object -filter { $_.Name -notlike  
'powershell*' } |  
Sort VM -descending | Select -first 10
```

如果你使用默认升序排序，你会想加入这最后的命令之前保留-last 10。

```
Get-Process | Where-Object -filter { $_.Name -notlike  
'powershell*' } |  
Sort VM -descending | Select -first 10 |  
Measure-Object -property VM -sum
```

如果这里使用的语法不合适，我们希望你至少能够找出最后一个Cmdlet的名称。

这个模型——运行一个命令、验证结果、键盘上的向上箭头键找回命令并修改再次尝试——就是PowerShell与传统脚本语言的区别。因为PowerShell是一个命令行Shell，可以立即返回结果，并且如果返回的结果不是期望结果，那么可以快速简单地修改命令。当你将已经掌握的少量的Cmdlets命令与刚学到的这一点结合后，你应该可以发现你所能够拥有的能力。

11.6 常见误区

在介绍Where-Object的时候，通常会遇到两个主要的困惑。我们试图在前面的讨论中涉及这些概念。但是如果你有任何疑问，将在这里得到解决。

11.6.1 请左过滤

你会希望你的过滤条件越接近开始的命令行越好。如果能在第一个Cmdlet后就完成过滤，那就这么做。如果不行，尝试在第二个Cmdlet命令后过滤，这样将尽可能减少后面Cmdlet命令的工作。

另外，尝试在尽可能靠近数据源的地方完成过滤。例如，你需要从一台远程计算机查询服务并使用Where-Object——正如本章的一个例子——考虑利用PowerShell的远程调用在远程计算机上进行过滤，这比把所有的对象都获取到本地之后再过滤要好得多。在第13章将会接触远程调用，并且会使用该方法重新过滤数据源。

11.6.2 何时允许使用\$_

特殊的\$_占位符只有在PowerShell知道如何寻找它时才有效。当它有效时，它一次只包含一个从管道传输到该Cmdlet命令的对象。请记住，不同的Cmdlet执行和产生结果的同时，在管道传输的生命周期中，管道中包含的内容也不断变化。

同样需要小心嵌套的管道——那些出现在括号里面的命令。例如，下面的示例可能会难以理解。

```
Get-Service -computername (Get-Content c:\names.txt |  
Where-Object -filter { $_ -notlike '*dc' }) |  
Where-Object -filter { $_.Status -eq 'Running' }
```

让我们慢慢梳理：

(1) 我们看到命令是以Get-Service开始的，但它却不是第一个执行的命令。这是因为圆括号内的Get-Content先执行。

(2) Get-Content 通过管道输出包含简单的String对象到Where-Object。Where-Object和过滤器处在圆括号内，\$_表示从Get-Content管道传输过来的String对象。只要字符串不是以“dc”结尾的，都会被保留并通过Where-Object输出。

(3) Where-Object的输出成为圆括号内的结果，因为Where-Object是圆括号内的最后一个Cmdlet命令。因此，所有不是以“dc”结尾的计算机名称会被发送到Get-Service的-computername参数中。

(4) 现在执行**Get-Service**，并且产生的**ServiceController**对象将会传输到**Where-Object**。该实例**Where-Object**会一次放置一个服务到**\$_**占位符，它会只保留那些属性为正在运行状态的服务。

有时候，我们觉得自己的眼睛会忽略所有的花括号、句号和圆括号，但是**PowerShell**就是这么工作的。而如果你能训练自己小心阅读命令，你将会理解命令做了些什么事情。

11.7 动手实验

注意： 对于本次动手实验来说，你需要运行**PowerShell v3**或更新版本**PowerShell**的计算机。

记住，不是只有**Where-Object**方式可以过滤，它甚至不应该是你第一个想到的命令。我们已经使得本章尽量保持简短，以便让你有更多的时间进行动手实验。所以，记住左过滤的原则，尝试完成下面的内容。

1. 导入**NetAdapter**模块（存在于最新版本的客户端或服务器版本的**Windows**中）。使用**Get-NetAdapter**命令显示一个非虚拟网络适配器列表（换言之，适配器的**Virtual**属性为**false**，也就是**PowerShell**的专用常量**\$False**）。

2. 导入**DnsClient**模块（存在于最新版本的客户端或服务器版本的**Windows**中）。使用**Get-DnsClientCache**命令显示一个从缓存中记录的**A**和**AAAA**列表。提示：如果你的缓存是空的，尝试浏览一些**Web**页面来强制一些项存入缓存中。

3. 显示一个关于安全方面的补丁列表。

4. 使用**Get-Service**是否可以显示一个自动启动类型且当前没有正在运行的服务列表？请仅仅回答是或者否。你不需要编写一个命令来完成该内容。

5. 显示一个管理员安装过的补丁列表，并列哪些是更新补丁。注意，有些补丁包没有“**installed by**”这个值，不过这没关系。

6. 显示名称为“Conhost”或“Svchost”且状态为“运行”的进程列表。

11.8 进一步学习

熟能生巧，所以尝试对你学习过的命令的输出内容进行过滤，比如Get-Hotfix、Get-EventLog、Get-Process、Get-Service甚至是Get-Command。例如，可以尝试对Get-Command的输出过滤，只剩下部分Cmdlet命令。或者使用Test-Connection来ping服务器，并且只有在没有应答的情况下显示结果。我们不建议你在每个实例中都使用Where-Object，但是你应该在适当的时候进行练习。

第12章 学以致用

是时候学以致用了。在本章，我们不会尝试教你任何新东西，而是使用你所学到的知识完成一个完整的示例。所以本示例必须是一个具有实际意义的示例。我们首先设定一个任务，在我们找出如何完成该任务之后，你可以跟随我们的进程。本章是本书内容的一个缩影，因为除了告诉你如何完成工作之外，我们还希望你意识到：你可以自学成才。

12.1 定义任务

首先，我们先假设你正在使用Windows 8或Windows Server 2012。很明显，这两个系统已经预装了PowerShell v3。如果你使用的Windows版本不是上面两个，如果可能，我们强烈推荐你下载一个试用版，或者使用例如CloudShare.com的服务开一个虚拟机。虽然PowerShell v3可以在一些老版本的Windows上运行，但这些版本并不像新版本那么深度管理集成。当然，使用更新版本的Windows或PowerShell也是可以的。

我们的目标是使用PowerShell创建一个计划任务。当我们创建完成后，可以在计划任务中看到该计划。该计划内容是每天凌晨3时将名称为“Accounting”的本地打印机中所有的作业删除。这么做是因为该打印机是一台老旧的设备，时不时会出现问题，导致作业无法完成。我们通过清理作业让每天有一个新的开始。

12.2 发现命令

完成任何任务的第一步都是找出哪一个命令可以完成任务。我们首先找出如何完成打印机相关工作。这样，我们就可以通过运行命令，确保找到的命令能完成我们希望做的工作。然后，我们将该命令放入计划任务，一次只解决一个问题。

我们首先开始寻找打印机相关的命令。注意我们选择*print*而不是*printer*作为关键字。如果可能，尽量使用单词更简短的形式，从而获得更多结果。

```
PS C:\> help *print*
```

Name	Category	Module
------	----------	--------

```

-----
Add-Printer          Function    printmanagement
Add-PrinterDriver    Function    printmanagement
Add-PrinterPort      Function    printmanagement
Get-PrintConfiguration Function    printmanagement
Get-Printer          Function    printmanagement
Get-PrinterDriver    Function    printmanagement
Get-PrinterPort      Function    printmanagement
Get-PrinterProperty  Function    printmanagement
Get-PrintJob         Function    printmanagement
Remove-Printer       Function    printmanagement
Remove-PrinterDriver Function    printmanagement
Remove-PrinterPort   Function    printmanagement
Remove-PrintJob      Function    printmanagement
Rename-Printer       Function    printmanagement
Restart-PrintJob     Function    printmanagement
Resume-PrintJob      Function    printmanagement
Set-PrintConfiguration Function    printmanagement
Set-Printer          Function    printmanagement
Set-PrinterProperty  Function    printmanagement
Suspend-PrintJob     Function    printmanagement
Out-Printer          Cmdlet     Microsoft.PowerShell.U

```

动手实验：你应该一步步跟随我们在本章运行的命令，从而确保你能看到我们所看到的结果以及信息。这里的重点不是完成任务，而是我们如何找出解决问题的方法。

上面的结果看起来有我们需要的命令，即**Get-PrintJob**和**Remove-PrintJob**。如果你查看**Remove-PrintJob**的帮助，你可以看到该命令有一个接受管道输入的**-InputObject**参数。这意味着我们可以获得作业对象，然后通过管道将该对象传递给**Remove-PrintJob**移除对象：

```

-InputObject <CimInstance#MSFT_PrintJob>

Required?          true
Position?          0
Accept pipeline input? true (ByValue)
Parameter set name  jobObject
Aliases           None
Dynamic?           False

```

在生产环境的打印机上尝试“**Get-PrintJob -printer "Accounting" | Remove-PrintJob**”后，确认该命令可以移除所有打印作业。因此，我们完成了打印机部分的工作。下面让我们开始计划任务部分。

```
PS C:\> help *task*
```

Name	Category	Module
-----	-----	-----
Disable-NetAdapterEncapsulated...	Function	NetAdapter
Enable-NetAdapterEncapsulatedP...	Function	NetAdapter
Get-NetAdapterEncapsulatedPack...	Function	NetAdapter
Set-NetAdapterEncapsulatedPack...	Function	NetAdapter
Get-CertificateNotificationTask	Cmdlet	PKI
New-CertificateNotificationTask	Cmdlet	PKI
Remove-CertificateNotification...	Cmdlet	PKI
Get-ClusteredScheduledTask	Function	ScheduledTasks
Get-ScheduledTask	Function	ScheduledTasks
Get-ScheduledTaskInfo	Function	ScheduledTasks
New-ScheduledTask	Function	ScheduledTasks
New-ScheduledTaskAction	Function	ScheduledTasks
New-ScheduledTaskPrincipal	Function	ScheduledTasks
New-ScheduledTaskSettingsSet	Function	ScheduledTasks
New-ScheduledTaskTrigger	Function	ScheduledTasks

很好——在一个名为**ScheduledTasks**的模块中发现了很多命令（当然，还有函数，但本质上是一回事）。现在让我们将搜索范围减少到仅该模块。

```
PS C:\> get-command -module scheduledtasks
```

Capability	Name
-----	-----
CIM	Get-ClusteredScheduledTask
CIM	Get-ScheduledTask
CIM	Get-ScheduledTaskInfo
CIM	New-ScheduledTask
CIM	New-ScheduledTaskAction
CIM	New-ScheduledTaskPrincipal
CIM	New-ScheduledTaskSettingsSet
Cmdlet, Script	New-ScheduledTaskTrigger
CIM	Register-ClusteredScheduledTask
CIM	Register-ScheduledTask
CIM	Set-ClusteredScheduledTask
CIM	Set-ScheduledTask
CIM	Start-ScheduledTask
CIM	Stop-ScheduledTask
CIM	Unregister-ClusteredScheduledTask
CIM	Unregister-ScheduledTask

很好，现在知道我们所需的是哪一个命令。剩下只需知道如何使用这些命令。

12.3 学习如何使用命令

我们希望创建一个新的计划任务，所以New-ScheduledTask看上去正是我们所需的命令。

```
PS C:\> help new-scheduledtask -full

NAME
    New-ScheduledTask

SYNTAX
    New-ScheduledTask [[-Action] <CimInstance#MSFT_TaskAction[]>]
    [[-Trigger] <CimInstance#MSFT_TaskTrigger[]>] [[-Settings]
    <CimInstance#MSFT_TaskSettings>] [[-Principal]
    <CimInstance#MSFT_TaskPrincipal>] [[-Description] <string>]
    [-CimSession <CimSession[]>] [-ThrottleLimit <int>] [-AsJob]
    [<CommonParameters>]
```

该命令不需要任何强制参数，但帮助显示-Trigger参数用于指定任务运行的时间，-Action参数用于指定所需完成的任务，其他参数基本上就可以忽略。帮助文档显示这两个参数接受的类型都为对象，即为TaskTrigger和TaskAction对象。所以我们需要找出如何生成这些对象。我们当然可以在帮助文档底部阅读示例代码，但这里我们小小挑战一下，因此不去阅读帮助。

通过查看ScheduledTask模块的任务列表，发现该列表中还包含了New-Scheduled TaskTrigger和New-ScheduledTaskAction命令，但愿可以找出我们所需要的。

```
PS C:\> help New-ScheduledTaskTrigger

NAME
    New-ScheduledTaskTrigger

SYNOPSIS
    New-ScheduledTaskTrigger [-RandomDelay <TimeSpan>] [-At]
    <DateTime>
    -Once [<CommonParameters>]

    New-ScheduledTaskTrigger [-DaysInterval <Int32>] [-RandomDelay
    <TimeSpan>] [-At] <DateTime> -Daily [<CommonParameters>]

    New-ScheduledTaskTrigger [-WeeksInterval <Int32>] [-RandomDelay
```

```
<TimeSpan>] [-At] <DateTime> -DaysOfWeek <DayOfWeek[]> -Weekly
[<CommonParameters>]

New-ScheduledTaskTrigger [-RandomDelay <TimeSpan>] [[-User]
<String>]
-AtLogOn [<CommonParameters>]

New-ScheduledTaskTrigger [[-RandomDelay] <TimeSpan>] -AtStartup
[<CommonParameters>]
```

当然，我们并不希望有任何随机的工作。第二个参数集包含强制的-Daily和-At参数，看上去正是我们所需要的。让我们试一试是否可以使用该命令创建一个触发器。

```
PS C:\> New-ScheduledTaskTrigger -daily -at 0300

WARNING: column "Enabled" does not fit into the display and was removed
Id      Frequency    Time              DaysOfWeek
--      -
0       Daily        1/1/0001 12:00:00 AM
```

不，上面的结果不太对，并不是我们想要的时间，因为结果是午夜。让我们再试一次：

```
PS C:\> New-ScheduledTaskTrigger -daily -at '3:00 am'

WARNING: column "Enabled" does not fit into the display and was
removed.
Id      Frequency    Time              DaysOfWeek
--      -
0       Daily        4/18/2012 3:00:00 AM
```

好的，该命令可以创建我们所需的触发器。非常好。注意前面两次创建的触发器的ID都为0，且该模块没有类似Get-ScheduledTaskTrigger的命令，这意味着PowerShell不会在内存中记录该对象。该命令产生一个触发器，但不会存储它。本例也说明了不要放过每一点信息（甚至是不起眼的ID），并多加留意看到的信息。

下面开始行动：

```
PS C:\> help New-ScheduledTaskAction
```



```
NAME
    New-ScheduledTaskAction

SYNTAX
    New-ScheduledTaskAction [-Execute] <string> [[-Argument] <string>]
    [[-WorkingDirectory] <string>] [-Id <string>] [-CimSession
    <CimSession[]>] [-ThrottleLimit <int>] [-AsJob]
    [<CommonParameters>]
```

我们已经在GUI中使用过计划任务，所以我们知道**-WorkingDirectory**用于设置任务运行所在的目录。**-Argument**可能用于将命令行参数传递给正在运行的任务，**-Execute**我们猜测用于执行希望运行的任务。让我们试一下：

```
PS C:\> New-ScheduledTaskAction -Execute 'dir'

Id                :
Arguments         :
Execute           : dir
WorkingDirectory  :
PSComputerName    :
```

貌似可行。让我们把命令连起来试一下：

```
PS C:\> New-ScheduledTask -Action (New-ScheduledTaskAction -Execute
'Get-PrinterJob -printer "Accounting"') -Trigger (New-ScheduledTaskTrigger -
daily -at '3:00 am') -Description "Reset accounting printer daily at 3am"
```

但是运行完上述命令后，我们无法在计划任务GUI（我们终于在“metro”风格的开始屏幕中找到了计划任务）看到该任务。郁闷，我们再次回到模块的命令列表：

```
PS C:\> gcm -Module scheduledtasks
Capability  Name
-----
CIM         Get-ClusteredScheduledTask
CIM         Get-ScheduledTask
CIM         Get-ScheduledTaskInfo
CIM         New-ScheduledTask
CIM         New-ScheduledTaskAction
CIM         New-ScheduledTaskPrincipal
```

```

CIM          New-ScheduledTaskSettingsSet
Cmdlet, Script New-ScheduledTaskTrigger
CIM          Register-ClusteredScheduledTask
CIM          Register-ScheduledTask
CIM          Set-ClusteredScheduledTask
CIM          Set-ScheduledTask
CIM          Start-ScheduledTask
CIM          Stop-ScheduledTask
CIM          Unregister-ClusteredScheduledTask
CIM          Unregister-ScheduledTask

```

嗯，**Register-ScheduledTask**可能是我们需要的，这看上去很有趣。以“**New**”开头的命令通常意味着创建某物，但我们还需要将新的任务“注册”（**register**），以便Windows能够得知该任务存在。

```

NAME
    Register-ScheduledTask

SYNTAX
    Register-ScheduledTask [-TaskName] <string> [[-TaskPath] <string>]
    [-Action] <CimInstance#MSFT_TaskAction[]> [[-Trigger]
    <CimInstance#MSFT_TaskTrigger[]>] [[-Settings]
    <CimInstance#MSFT_TaskSettings>] [[-User] <string>] [[-Password]
    <string>] [[-RunLevel] <RunLevelEnum> {Limited | Highest}]
    [[-Description] <string>] [-Force] [-CimSession <CimSession[]>]
    [-ThrottleLimit <int>] [-AsJob] [<CommonParameters>]

```

看上去和**New-ScheduledTask**非常类似，只是参数更多。如**-TaskName**参数，我们推测该参数用于给任务赋予名称。我们还看到**-User**和**-Password**，这两个参数可以在计划任务中看到。好了，让我们尝试下面命令：

```

PS C:\> Register-ScheduledTask -TaskName "ResetAccountingPrinter" -
Descript
ion "Resets the Accounting print queue at 3am daily" -Action (New-
Scheduled
TaskAction -Execute 'Get-PrintJob -printer "Accounting"') -Trigger
(New-Sch
eduledTaskTrigger -daily -at '3:00 am')

WARNING: column "State" does not fit into the display and was removed.
TaskPath          TaskName
-----
\                  ResetAccountingPrinter

```



图12.1 在计划任务的GUI中确认计划任务是否存在

看上去该命令完成了某些操作。回到GUI，如图12.1所示，此时刚创建的任务存在了。太棒了！抱歉，当难题终于解决时，我们有点小激动。

很好，现在我们感觉就像超级英雄。但重点并不是我们完成了该项工作——而是发现如何完成工作。当然，我们也完成了这一点。当我们写本章时，我们故意选择一个别人说不可能完成且我们没有经历过的任务。因此在跟随我们学习和探索的过程中，每一点探索都可能伴随报错。

12.4 自学的一些技巧

再次说明，本书的目的是教会你如何自学——这也是本章希望阐明的。下面是一些技巧：

- 不要害怕使用帮助并确保阅读示例。我们不止一次强调过这一点，但好像没人愿意听。我们仍然会看到很多学生在我们眼皮底下使用Google寻找示例。为什么那么害怕帮助文档？如果你都愿意读别人的博客了，为什么不先尝试在帮助文档中阅读示例？

- 请注意，在屏幕上，每一点信息可能都非常重要——请不要跳过不是你目前正在寻找的信息。你很容易这样做，但请不要这么做。要查看每一部分信息，并尝试发现该信息的用处，以及能使用该信息能够推算出什么。当每次创建触发器的ID都为0，而不是每次创建触发器都有一个连续递增的触发器时，我们可以安全地确认PowerShell不会将该触发器存到某个列表。这还意味着我们需要将触发器传递给某个父命令，而不是先创建它供后续使用。
- 不要害怕失败，希望你弄一个虚机，然后在虚拟里玩PowerShell。学生们经常会问类似这类问题：“如果我做了这个和那个，会发生什么？”我们的回答往往是“不知道，自己试试”。在虚拟机做实验是一个好办法，最坏的情况也只不过是回滚到某个快照点，对吧？所以无论做什么，都请试一试。
- 如果尝试一种方法不奏效，不要挠墙——请尝试其他方法。本例中我们指定了错误的时间格式0300，且懒得去读示例中的正确写法。我们选择了将'3:00 am'作为字符串，而不是不停尝试03:00、03:00:00等格式。
- 我们利用对GUI计划任务的直觉猜出一些命令，比如-Execute开关。请不要让使用命令行Shell导致你忘记过去使用Windows的经验——请尝试将你所做的和你过去所做的工作产生关联。

很明显，随着时间的流逝，所有的事情都会变得简单。请耐心并保持练习——但同时在学习过程中不忘思考。

12.5 动手实验

注意： 对于本次动手实验来说，你需要运行PowerShell v3或更新版本PowerShell的计算机。

下面该轮到你了。我们假设你正在使用虚拟机或其他你可以假借学习的名义搞乱的环境。请不要在生产环境和运行关键系统的计算机上进行实验。

Windows 8和Windows Server 2012包含一个使用文件共享的模块。你的任务是创建一个名称为“LABS”的目录，并共享该目录。为了练习的简便，先假设该目录和共享不存在。先不用管NTFS的权限问题，但请确保共享目录的权限设置为所有人拥有读/写权限，并且本地管理员拥有完全控制权。由于共享的主要是文件，你还希望为文档设置共享缓存。你的脚本还应该展示新建的共享及其权限。

第13章 远程处理：一对一及一对多

当首次使用PowerShell（第一版）时，我们最先接触的是Get-Service命令，检查后发现该命令包含一个-ComputerName参数。这是否意味着它也可以读取其他计算机上的服务名称？经过一些简单的测试之后，我们发现的确如此。我们感到很惊喜，同时也查看其他有-ComputerName参数的Cmdlet。令人失望的是，包含该参数的Cmdlet非常少。但在第二版PowerShell（新增一系列的命令）后，包含-ComputerName参数的命令数目远远多于不包含该参数的命令数目。

从那时起，我们意识到PowerShell的开发有点懒惰——其实，这是好事。因为他们不希望每个Cmdlet都带上-Computer参数，所以他们创建了一个Shell级别的系统，命名为远程处理（Remoting）。该系统使得你可以在一个远程计算机上运行任何Cmdlet。甚至当本地计算机没有包含某些命令时，你也可以直接运行远程计算机上已存在的这些命令（也就是说，你们不需要在本地计算机上安装任何一个管理性质的Cmdlet）。远程处理系统的功能非常强大，它提供了大量有趣的管理功能。

注意： 远程处理是非常庞大和复杂的一门技术。在本章中，我们会介绍该项技术，同时会覆盖到日常工作中大概百分之八十到九十的场景。正因为我们没有覆盖到全部知识点，所以在本章最后的“进一步探讨”小节中，我们会列出一些学习远程处理全部配置选项的资源。

13.1 PowerShell远程处理的原理

在一定程度上讲，PowerShell的远程处理类似于Telnet或者其他一些老旧的远程处理技术。当你键入该命令时，它会在远程计算机上运行。只有该命令的执行结果会返回本地计算机。和Telnet和SSH不一样的是，PowerShell采用一种新的通信协议，我们称之为针对管理的Web服务（Web Services for Management，WS-MAN）。

WS-MAN完全通过HTTP或者HTTPS进行工作，这样保证必要的情况下，能轻易透过防火墙进行作业（因为每种协议都使用唯一的端口进行通信）。微软对WS-MAN的实现主要基于一个后台服务：Windows远程管理组件（WinRM）。在安装第二版PowerShell的时候会同时安装WinRM，在服务器版操作系统（比如Windows Server 2008 R2）中默认开启该服务。Windows 7操作系统默认安装该服务，但是该服务处于禁用状态。从Windows Server 2012开始，WinRM服务集成在第三版PowerShell中，默认开启状态。

到目前为止，你已经知道Windows PowerShell的Cmdlet会产生一些对象作为输出结果。当你执行一个远程命令时，它会将输出结果放入一个特定形式的包中，之后通过网络中的HTTP（或者HTTPS）协议传回本地计算机。XML已经被证明是针对该问题的优秀解决方案，所以PowerShell会将输出对象序列化到XML中。下一步，XML文件会通过网络进行传输。当到达本地计算机之后，该XML会反序列化为PowerShell可以处理的对象。序列化和反序列化仅仅是一种格式转换的形式：从对象转化为XML称为序列化，从XML转为对象则为反序列化。

为什么你需要关注输出结果返回的方式？因为这些序列化和反序列化的对象只是各种快照而已，它们并不会随着后续状态的变化而自我更新。例如，如果你获得代表远程计算机上运行进程的对象时，这些对象只能反映对象被产生时刻的状态。例如，对象包含的内存使用数据或者CPU使用率数据并不会随着后续的变化而变化。另外，你无法对反序列化对象下达任何指令（例如，你无法下达停止的指令）。

这些都是针对远程处理的一些比较基本的限制，但是却无法阻止操作者通过其他方法来实现一些神奇的功能。实际上，完全可以下达指令让远程处理自行停止，但是我们需要更加聪明一些。本章后续部分会展示该场景。

要保证远程处理可以正常工作，需要满足下面两个条件。

- 本机计算机和远程计算机（需要运行命令的计算机）至少需要第二版或者更新版本的PowerShell。Windows XP是第二版PowerShell支持的最老版本操作系统，所以它也是能实现远程处理功能最老版本的操作系统。
- 理论上，两台计算机需要在同一域或者可信任的域中。如果计算机不在域中，远程处理也可以正常工作，但配置会稍微麻烦。本章中不会讲到这一点。如果你想了解该类场景，请在PowerShell中执行Help About_Remote_Troubleshooting。

动手实验： 希望你可以模拟本章中的示例。为了完成这些实验，理论上，你需要第二台计算机（当然，也可以是一个虚拟机），并且这两台计算机需要在同一个域中。远程计算机可以运行在已经安装第二版或者更新版本PowerShell的任意操作系统上。当然，如果无法找到第二台计算机或者虚拟机，也可以使用localhost来创建到当前计算机的伪远程连接，但是无法像真实远程处理远程计算机那样让人激动兴奋。

13.2 WinRM概述

首先我们会讲到WinRM，因为在使用远程处理之前，我们必须配置该服务。再次申明，你只需要在接收远程命令的计算机上配置WinRM以及PowerShell远程处理即可。在我们大部分工作环境中，Windows管理员都会开启每台Windows环境计算机上的远程服务（请记住，从Windows XP开始，所有操作系统均支持PowerShell和远程服务）。这样做能保证你可以使用后台远程连接到客户端计算机或者笔记本电脑（也就是说，这些计算机的用户根本不知道你有远程连接到该计算机）。对我们来说，该项功能非常有用。

并非仅有PowerShell能使用WinRM服务。实际上，微软在越来越多的管理程序中开始使用WinRM服务——甚至包含已经使用了其他协议的那些程序。基于这一思想，微软保证WinRM可以将流量导入至多种管理程序——不仅仅是PowerShell。WinRM类似一个调度器：当有新的流量进来后，WinRM会决定由哪种程序来处理这部分流量。所有WinRM流量都标记了接收应用程序的名称，同时这些应用程序都必须在WinRM中创建各自的端点，这样WinRM才能侦听这些主体的流量。也就意味着，你们不只需要启用WinRM服务，也需要在WinRM中将PowerShell注册为一个端点。图13.1说明了这些组件如何组合在一起。

如图13.1所示，在你的系统中可以有几十个甚至上百个WinRM端点（PowerShell称它们为会话配置选项）。每一个端点都指向一种应用程序，甚至你可以将多个端点指向同一个应用程序，但是每个端点提供不同的权限以及功能。例如，你可以在环境中创建一个PowerShell端点，该端点仅允许特定用户执行一个或者两个命令。在本章中不会深入讲解远程处理，但是我们会在第23章中再深入探讨该服务。

图13.1也阐述了WinRM侦听器部分，在图中属于HTTP种类中的一种。一个侦听器会为WinRM等待网络流量的进入——有点像Web服务器侦听传入请求。尽管由Enable-PSRemoting创建的默认侦听器会侦听本地所有IP地址的某个端口，但是一个侦听器仅会侦听从特定IP地址的特定端口发出的请求。

侦听器会连接到已定义的端点。我们可以采用下面的方法创建一个端点：新开一个PowerShell窗口——需要以管理员权限运行该命令，之后执行Enable-PSRemoting命令。有些时候你可能会看到另外一个相关的命令Set-WSManQuickConfig。但是根本没必要手动运行该命令，Enable-PSRemoting命令会自动调用该命令。另外，Enable-PSRemoting命令也会执行其他一些步骤来完成开启远程处理服务。总体来说，该Cmdlet会开启WinRM服务，

配置该服务为自动启动模式，然后在WinRM中为PowerShell注册一个端口，甚至会在Windows防火墙中针对传入的WinRM流量创建例外条件。

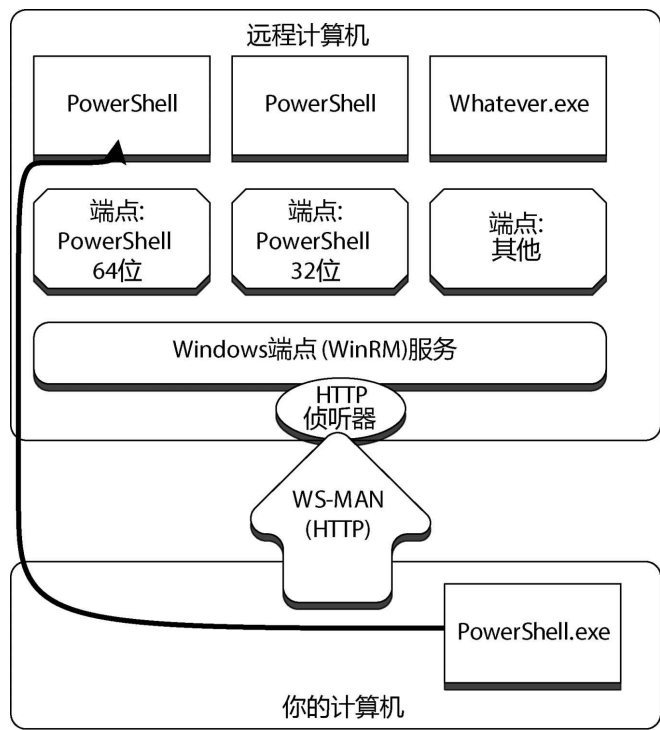


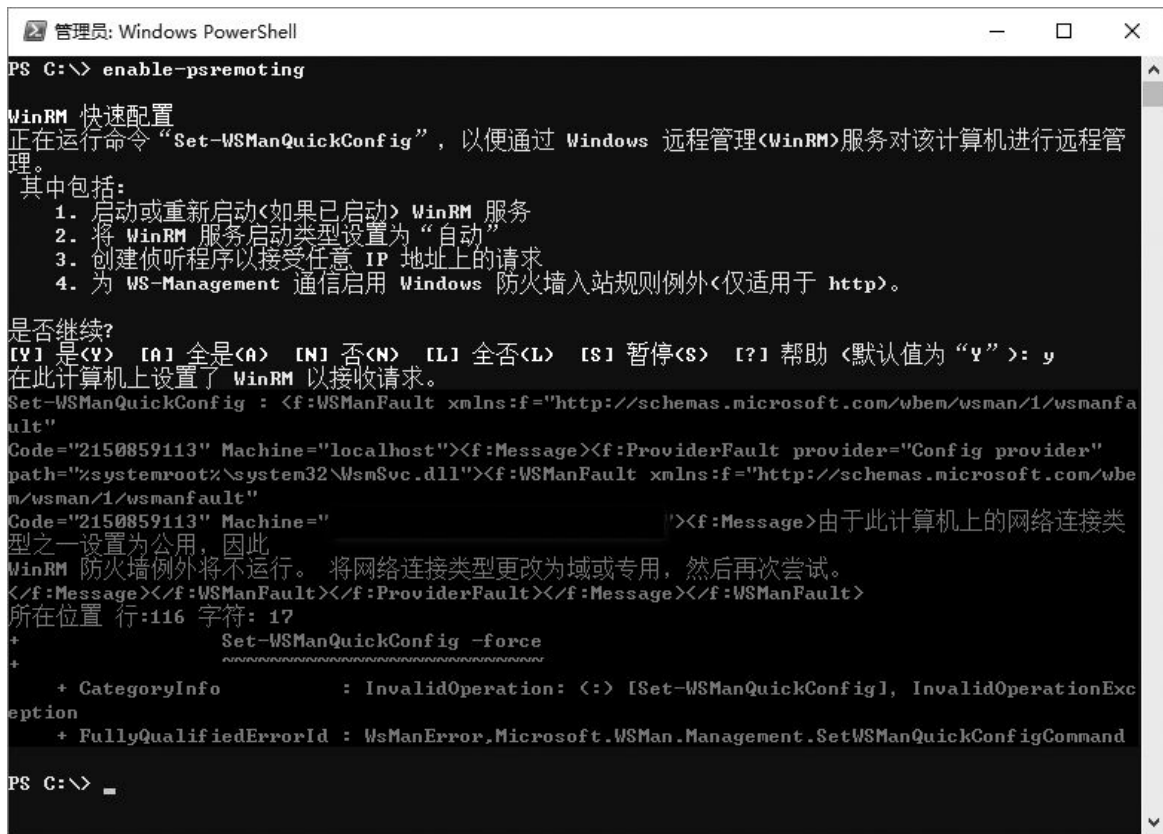
图13.1 WinRM、WS-MAN、端点和PowerShell之间的关系

动手实验： 该实验环节需要你在第二台计算机（如果你只有一台计算机的话，那么请在当前计算机启用）上启用远程处理服务。请确保你是在管理员权限下运行PowerShell（PowerShell的窗口边框上显示了“管理员”字样）。如果不是，那么请关闭当前窗口，然后右键单击开始菜单中的PowerShell按钮，选择“以管理员身份运行”。如果在启用远程处理时返回了一些错误信息，那么请暂停并解决该问题。只有当Enable-PSRemoting正确无误运行时，才能继续后面的步骤。

图13.2中显示了当运行Enable-PSRemoting命令时经常会遇到的一个错误。

图13.2中的错误一般只会出现在客户端计算机上，并且如果你深入研究这个错误信息，你可以看到导致这个错误的原因。我们设置至少一个网卡为“公用网络”类型。请记住，在Windows Vista以及之后版本的操作系统中，对每一个网卡都会设置一个网络类型（家庭网络、工作网络或者公用网络）。类型为“公用网络”的网卡中无法设置Windows防火墙例外，所以当我们在执行Enable-PSRemoting命令尝试创建一个防火墙例外时，就会失败。唯一的解决办法是进入Windows中，修改该网卡的类型为“工作网络”或

者“家庭网络”。但是，如果你是连接到一个公用网络（比如一个公用的网络热点），请不要这样做，因为这将关闭一些重要的安全保护功能。



```
管理员: Windows PowerShell
PS C:\> enable-psremoting

WinRM 快速配置
正在运行命令“Set-WSManQuickConfig”，以便通过 Windows 远程管理(WinRM)服务对该计算机进行远程管理。
其中包括：
1. 启动或重新启动(如果已启动) WinRM 服务
2. 将 WinRM 服务启动类型设置为“自动”
3. 创建侦听程序以接受任意 IP 地址上的请求
4. 为 WS-Management 通信启用 Windows 防火墙入站规则例外(仅适用于 http)。

是否继续?
[Y] 是(Y) [A] 全是(A) [N] 否(N) [L] 全否(L) [S] 暂停(S) [?] 帮助 (默认值为“Y”)： y
在此计算机上设置了 WinRM 以接收请求。
Set-WSManQuickConfig : <f:WSManFault xmlns:f="http://schemas.microsoft.com/wbem/wsman/1/wsmanfault"
Code="2150859113" Machine="localhost"><f:Message><f:ProviderFault provider="Config provider"
path="%systemroot%\system32\WsmSvc.dll"><f:WSManFault xmlns:f="http://schemas.microsoft.com/wbem/wsman/1/wsmanfault"
Code="2150859113" Machine="localhost"><f:Message>由于此计算机上的网络连接类型之一设置为公用，因此 WinRM 防火墙例外将不运行。 将网络连接类型更改为域或专用，然后再次尝试。
</f:Message></f:WSManFault></f:ProviderFault></f:Message></f:WSManFault>
所在位置 行:116 字符: 17
+ ~~~~~
+ Set-WSManQuickConfig -force
+ ~~~~~
+ CategoryInfo          : InvalidOperation: (:) [Set-WSManQuickConfig], InvalidOperationException
+ FullyQualifiedErrorId : WsManError,Microsoft.WSMan.Management.SetWSManQuickConfigCommand

PS C:\>
```

图13.2 在客户端计算机启用远程处理时容易出现的一个错误信息

注意：如果运行的是服务器版的操作系统，你没必要担心这个错误，因为在该版本操作系统中，并没有这个限制。

如果你对需要到每台计算机上去开启远程处理服务感到很厌烦，没关系，你也可以通过组策略对象（GPO）来实现。这些必要的GPO设置选项已经内置到Windows Server 2008 R2（以及后续版本操作系统）的域控制器计算机中（如果是老版本操作系统的域控制器计算机，那么需要去网站<http://download.microsoft.com>上下载一个ADM（Administrative Templates）模板，之后添加这些GPO选项）。打开一个GPO对象，之后查看路径“计算机配置”>>“管理模板”>>“Windows组件”下的对象。在该列表的中间部分，你可以看到“Windows远程解释器”（Windows Remote Shell）和“Windows远程管理”（WinRM）。现在，我们假定你将会在希望配置的那些计算机上运行Enable-PSRemoting命令，因为当前你可能只有一台或者两台虚拟机。

注意：PowerShell的About_Remote_TroubleShooting帮助主题中包含了更多关于如何使用GPO对象的内容。你可以查找该帮助信息中的“如何启用企业中的远程处理”和“如何通过使用组策

略启用侦听器”部分。

第二版的WinRM服务（第二版以及后续版本的PowerShell使用的WinRM版本）默认会使用TCP端口5985来侦听HTTP，使用5986端口来侦听HTTPS。这样的端口号保证不会与本地安装的任意Web服务器使用的端口号（一般使用80~443之间的端口号）相冲突。使用Enable-PSRemoting创建的远程处理默认仅对5985端口创建非加密的HTTP侦听器。当然，你也可以配置WinRM使用其他端口，但是我们不建议这样做。如果我们采用默认值，所有的PowerShell远程处理命令都可以正常执行。假如我们修改了端口号，在我们输入远程处理命令时，我们必须指定端口号，这样也就意味着我们键入更多的字符。

如果你确定要修改端口号，可以通过下面的命令实现。

```
WinRM Set WinRM/Config/Listener?Address=*+Transport=HTTP  
→@{Port="1234"}
```

在该示例中，1234是你希望使用的端口号。如果将其中的HTTP修改为HTTPS，则该命令可用作修改HTTPS的侦听端口。

别动手实验：尽管在生产环境中可能需要修改该端口，但是请不要在测试计算机上进行修改。保留WinRM默认配置选项，这样本章后面的示例才可以正常执行（不需要你再做额外的修改）。

其实存在另外一个方法，可以去修改客户端计算机上的WinRM的默认端口。这样当我们在执行命令的时候，就不需要再指定修改之后的默认端口。但是在本书中，我们仍然使用微软提供的默认配置选项。同时，我们也会提示你可以针对WinRM创建多个侦听器（比如一个针对HTTP流量，一个针对加密的HTTPS流量，或者其他一些针对不同的IP地址）。这些侦听器会将流量导入至计算机上配置的端点。

注意： 如果你有查看Windows远程解释器（Remote Shell）的组策略对象设置选项，你或许注意到下面几个可更改的选项：一个远程处理进程在被计算机自动杀掉之前处于打开状态的最长时间；允许并行执行远程处理进程的最大用户数；每个远程解释器可使用的最大内存以及最大进程数；每个用户可打开远程解释器的最大数目等。针对健忘的管理员来说，这些配置选项都是确保服务器不会过度消耗资源很好的方法。默认情况下，只有管理员才能使用远程处理，所以没有必要担心普通用户会导致服务器资源用尽。

13.3 一对一场景的Enter-PSSession和Exit-PSSession

PowerShell可以通过两种方法实现远程处理，第一种称为一对一或者1:1远程处理（第二种称为一对多，或者1:n远程处理，在下一节中会讲到一对多场景）。当使用一对一远程处理时，实际上是在单台远程计算机上调用了一个Shell命令窗口。输入的任何命令都会直接在该计算机上运行，然后在远程处理窗口中返回输出结果。该机制非常类似于远程桌面连接（Remote Desktop Connection），只是Windows PowerShell是采用命令行环境。相对于远程桌面连接，这种远程处理技术只需要使用很少的资源，所以对服务器来讲，开销会小很多。

如果需要针对一台远程计算机建立一对一的远程处理进程，请执行下面的命令：

```
Enter-PSSession -ComputerName Server-R2
```

（你需要使用正确的计算机名称来替代Server-R2。）

假如在远程计算机上已经启用了远程处理，两台计算机在同一个域中，并且网络质量良好，那么你应该可以得到一个连接。如果Shell命令窗口变为下面的格式，那么也就说明该连接成功建立。

```
[Sever-R2] PS C:\>
```

该Shell命令框表示你所执行的任何语句都是在Server-R2（或者说是你连接到的计算机）上运行。你可以在该命令框中运行任意命令，甚至可以在远程计算机上导入已存在的任意模块或者添加任意PSSnapIn。

动手实验：现在请尝试建立一个到第二台计算机或者虚拟机的远程连接。如果你从来没有这样测试过，那么你需要在尝试连接远程计算机之前，在该机器上启用远程处理功能。另外要注意的是，你需要知道远程计算机的真实名称，WinRM默认不允许使用IP地址或者DNS中的别名去进行远程处理。

你的权限以及特权在远程连接中也会继续保持。你运行的PowerShell副本会带有其运行的安全令牌（该过程通过Kerberos实现，所以并不会通过网络传递用户名以及密码到远程计算机）。你在远程计算机上执行的任何命令都依赖于你的凭据，所以你能实现你权限范围内的任意操作。该过程类似于你通过远程桌面连接到对应的远程计算机，然后在该计算机上执行本地的PowerShell。

下面介绍两个不同点：

- 即使远程计算机上PowerShell存在一个Profile脚本，当使用远程处理时，该脚本也不会运行。我们还没有讲到Profile脚本部分（在25章中会涉及该部分知识），但是这里可以大概提一下，Profile脚本是指当我们打开Shell时会自动运行的一批命令。人们经常使用Profile脚本来自动载入一些Shell扩展程序以及模块等。我们必须明白，当我们通过远程处理连接到某台计算机时，对应的Profile脚本不会自动执行。
- 远程计算机的执行策略会限制某些脚本的运行。假如本地计算机的策略设置为RemoteSigned，也就意味着可以运行本地未签名的脚本。如果远程计算机的策略设置为默认（严格），当使用远程处理连接到该计算机时，并不是所有脚本都可以运行。

了解这两个不同之处之后，可以继续后面的学习了。但是等等——当在远程计算机上执行命令结束之后，还要执行什么命令呢？很多PowerShell的Cmdlet都是以成对形式出现，一个Cmdlet做一件事情，另一个Cmdlet就会做相反的事情。在这个场景中，如果Enter-PSSession可以对其他计算机执行远程处理，你能猜到什么命令可以退出该进程吗？如果你猜到是Exit-PSSession，那么请给你自己一个奖励。该命令不需要其他任何参数；执行之后，Shell命令窗口会变回正常，远程连接会被自动关闭。

动手实验： 如果已经开启远程处理进程，执行Exit-PSSession命令并退出远程连接进程。

如果你忘记执行Exit-PSSession而是直接关闭PowerShell的窗口，会有什么后果呢？别担心。PowerShell和WinRM足够智能，它们能识别你的行为，然后自行关闭远程连接。

有个要点需要注意：当你使用远程处理连接到另一台计算机时，除非完全理解你所做的操作，否则不要在该命令窗口中再次执行Enter-PSSession。假如你的本地计算机为Computer A，使用Windows 7操作系统，之后使用远程处理连接到Server-R2。此时，在PowerShell命令框中，执行下面的语句：

```
[Server-R2] PS C:\>Enter-PSSession Server-DC4
```

该语句会在Server-R2上维护一个到Server-DC4的远程连接，也就是会建立一个“远程处理链”。该链非常难以追踪，同时会增加计算机中不必要的系统开销。在某些场景下，可能只能采取这种方式来实现——比如

Server-DC4处于防火墙中，无法被直接访问，所以需要Server-R2作为中转服务器，使得可以访问到Server-DC4。但是，一般情况下，请不要使用远程处理链。

警告： 在某些人看来，远程处理链类似于为“二连跳”，同时可以算作PowerShell的一个缺点。简单提示一下：如果PowerShell的命令行窗口已经显示了一个计算机的名称，那么请到此结束。除非你退出该进程回到本地计算机命令（PowerShell命令行窗口中不包含计算机名称）时，你才能再次执行一些远程控制的命令。我们会在第23章中讨论启用多层远程处理的相关问题。

当你使用一对一的远程处理时，你不需要担心被序列化和反序列化的对象。就你个人而言，其实等效于直接在远程计算机的控制台中键入命令。如果你获取了一个进程，并通过管道传递给Stop-Process命令，正如我们期待的那样，该进程会停止运行。

13.4 一对多场景的Invoke-Command

下面讲的是Windows PowerShell最酷的功能之一，也就是将一个命令同时传递给多台远程计算机。是的，就是这样，也可称之为全面的分布式计算。每台计算机都独立执行发送的命令，然后将结果集返回给你。

PowerShell利用Invoke-Command命令来实现该功能，称之为一对多或者1:n远程处理。

该命令类似下面的语句：

```
Invoke-Command -ComputerName Server-R2,Server-DC4,Server12  
➔-Command {Get-EventLog Security -Newest 200 |  
➔Where {$_.EventID -eq 1212 }}
```

动手实验： 尝试运行上面的脚本，请使用你自己的远程计算机的名字来替换上面的三个计算机名称。

最外层大括号{}中包含的全部命令都会传递到三台远程计算机。默认情况下，PowerShell最多一次与32台远程计算机通信。如果超过32台，那么会将计算机信息存放到一个队列中。如果命令在一台远程计算机上运行完毕，队列中的下一台计算机立即开始运行。当然，如果网络足够良好，并且计算机足够强劲，那么我们可以通过Invoke-Command的-ThrottleLimit参数来指定更多数量的计算机（如果需要了解更多的信息，请查阅该命令的帮助文档）。

注意标点符号

我们需要注意一对多远程处理示例的语法，因为PowerShell的标点符号在某种场景下会让我们感到很困惑。当我们在输入这些命令时，这些困惑可能让PowerShell实现一些意料之外的功能。

比如，考虑下面的示例：

```
Invoke-Command -ComputerName Server-R2,Server-DC4,Server12  
➔-Command {Get-EventLog Security -Newest 200 |  
➔Where { $_.EventID -EQ 1212 }}
```

在该示例中，有两个命令使用了大括号：Invoke-Command和Where（是Where-Object命令的别名）。Where命令完整嵌套在最外层的大括号中。最外层的大括号中包含的命令就是我们需要传递到远程计算机上执行的命令，也就是下面这段命令。

```
Get-EventLog Security -Newest 200 | Where { $_.EventID -EQ 1212 }
```

可能很难去照搬这些嵌套的命令，特别是本书中这种示例，由于每页的宽度限制，必须使用多行文字来展示该命令。

你必须确保你知道传递给远程计算机的命令到底是什么，同时要理解每一组大括号的功能。

另外，需要告知你的是，在Invoke-Command的帮助信息中是找不到-Command参数的，但是我们确认上面示例中的命令可以正常执行。实际上，-Command参数是帮助文档中-ScriptBlock参数（可以在Invoke-Command帮助文档中找到该参数的信息）的一个别名或者昵称。由于-Command命令更容易记住，所以我们往往使用-Command，而不会使用-ScriptCommand。但是实际上，它们的作用相同。

如果你认真查阅了Invoke-Command的帮助文档，你应该会注意到其中一个参数，该参数允许我们指定一个脚本文件，而不是一个命令。该参数可以将本地的完整脚本传递到远程计算机——意味着你可以自动化一些复杂的任务，让每一台计算机完成各自对应的部分。

动手实验： 确保你在Invoke-Command的帮助文档中找到了-ScriptBlock参数，同时能找到允许指定一个文件路径以及名称的参数（-FilePath）（并不是一段脚本）。

现在让我们回到本章开始提到的-ComputerName参数。当我们首次使用Invoke-Command时，我们键入了一串以逗号分隔的计算机名称，比如前面的示例。但是真实环境中可能存在大量的计算机，因此我们并不想每次都手动键入这些计算机名称。我们可以将所有的计算机按照对应的种类，比如Web服务器和域控制器服务器，放入到一个文本文档中。文本文档的每行代表一个计算机名称——不需要使用逗号，引号。通过PowerShell，我们可以很轻易地使用这些文本文档中的内容：

```
Invoke-Command -Command {dir}
➔-ComputerName (Get-Content WebServers.txt)
```

上面命令中的圆括号使得PowerShell优先执行Get-Content命令——和数学中的圆括号功能一样。之后Get-命令的结果集被传递给-ComputerName参数，然后括号中的命令就可以在文件中罗列的计算机上运行。

有些时候我们会遇到更棘手的问题，比如从活动目录中获取计算机的名称。我们可以使用Get-ADComputer命令（来自于活动目录模块；Windows 7的远程服务器管理工具（RSAT）/Windows Server 2008 R2或者之后版本的操作系统的域控制器服务器中均存在该模块）来获取计算机信息，但是我们无法将该命令放入圆括号中（类似上面的Get-Content命令）。为什么不行呢？因为Get-Content命令产生的对象类型为-Computer参数可接受的简单文本String类型。但是Get-ADComputer会输出完整的计算机对象，-ComputerName参数不知道应该如何处理这部分数据。

如果我们要使用Get-ADComputer命令，那么我们需要找到一个方法去获取这些计算机对象名称属性的值。比如下面的命令：

```
Invoke-Command -Command {dir} -ComputerName (
➔Get-ADComputer -Filter * -SearchBase "OU=Sales,DC=Company,DC=pri" |
➔Select-Object -Expand Name)
```

动手实验：如果你是在Windows Server 2008 R2的域控制器服务器或者安装了远程服务器管理工具（RSAT）的Windows 7计算机上运行PowerShell，那么你可以执行Import-Module ActiveDirectory，之后再运行上面的命令。如果你的测试域环境中并没有包含计算机账户的Sales OU，那么你需要将OU=Sales修改为OU=Domain Controllers，同时需要根据你本地实际域情况修改Company和Pri为正确的值（比如，你的域

名为mycompany.org，那么你需要使用mycompany替换company，使用org替换pri）。

通过使用圆括号，我们将产生的计算机对象通过管道传递给Select-Object，然后选择其中的-Expand参数。我们会告诉该命令去获取对应值的Name属性——在这个示例中，也就是这些计算机对象。圆括号中命令的执行结果是一串计算机名称，而不是计算机对象，正好-ComputerName参数能处理的对象就是计算机名称。

注意： 我们希望前面对-Expand参数的讨论能让你有种似曾相识的感觉：你应该是在第9章中第一次看到该参数。如果需要，请回到第9章对应小节以便加深印象。

如果需要深入了解，那么我们需要查看每个参数代表的意义。Get-ADComputer的-Filter参数指定所有的计算机都应该包含在这个命令的输出列表中；-SearchBase参数指定我们要从哪个地方开始执行这个命令——在这个示例中，是指Company.Pri域的Sales OU。再次说明，Get-ADComputer仅在Windows Server 2008 R2（及之后版本操作系统）的域控制器服务器上以及安装远程服务器管理工具（RSAT）Windows 7（及之后版本操作系统）的客户端电脑上才存在。

13.5 远程命令和本地命令之间的差异

在这里，我们会解释使用Invoke-Command命令远程运行和在本地运行相同命令之间的差异，也会涉及使用远程处理以及使用其他形式远程连接之间的差异。我们会使用下面的命令作为演示差异的示例。

```
Invoke-Command -ComputerName Server-R2,Server-DC4,Server12  
➔-Command {Get-EventLog Security -Newest 200 |  
➔Where {$_.EventID -EQ 1212}}
```

之后我们再看一些其他命令，然后确认为什么它们会不一样。

13.5.1 Invoke-Command VS -ComputerName

下面是实现相同目的的另外一种方法。

```
Get-EventLog Security-Newest 200  
➔-ComputerName Server-R2,Server-DC4,Server12 |
```

```
➔Where {$_.EventID -EQ 1212}
```

在该示例中，我们使用了Get-EventLog命令的-ComputerName参数，而不是远程调用整个命令。我们会得到差不多相同的结果，但是在该命令执行的方式上存在很大的不同。

- 提及的计算机将按照顺序被串行访问，并不会采用并行方式，也就意味着命令会执行更久的时间。
- 该命令的输出结果中不会包含PSComputerName属性，也就意味着我们很难判别某个结果是从哪台计算机得出的。
- 该连接并不会使用WinRM来实现，而会使用.Net Framework决定的底层协议。我们不知道到底是哪种协议，同时有可能由于该协议无法在本地和远程计算机之间顺利通过防火墙而无法建立连接。
- 我们会从3台计算机上查询200条记录，然后通过它们找到eventid为1212的事件。也就意味着，可能会返回一些我们不需要的结果。
- 我们得到的是功能全面的事件日志对象。

对带有-ComputerName参数的Cmdlet而言都存在这些差异。一般来讲，使用Invoke-Command命令比Cmdlet的-ComputerName参数更有效率，更有用。

如果我们采用之前的Invoke-Command命令，就会是下面这样：

- 计算机会被并发地访问，也就意味着，命令执行更有效率。
- 命令的输出结果中包含PSComputerName属性，也就使得我们能轻易看到哪个结果来自于哪台计算机。
- 通过WinRM来建立连接，WinRM会使用一个预定义的端口，使得可以更轻易地穿过任何防火墙。
- 每台计算机都会查询200条记录，然后在本地就做筛选。通过网络传递回来的数据是经过筛选之后的结果，也就是说，这些记录都是我们希望得到的有效数据。
- 在传递结果之前，每台计算机都会将输出结果序列化为XML。本地计算机收到该XML之后，会反序列化为一些类似对象的结果。但是它们并不是真正的事件日志对象，这也就限制了本地计算机处理这些对象的方式。

最后一点是使用-ComputerName参数和Invoke-Command命令之间很大的一个差异点。下面会讨论到这一点。

13.5.2 本地处理VS远程处理

再次引用之前的示例:

```
Invoke-Command-ComputerName Server-R2,Server-DC4,Server12  
➔-Command {Get-EventLog Security -Newest 200 |  
➔Where {$_.EventID -EQ 1212}}
```

然后和下面的命令对比一下:

```
Invoke-Command -ComputerName Server-R2,Server-DC4,Server12  
➔-Command {Get-EventLog Security -Newest 200 } |  
➔Where {$_.EventID -EQ 1212}
```

看起来差异很小。唯一的不同点是,我们移动了一个大括号的位置。

在第二个命令中,只有**Get-EventLog**命令被远程调用。**Get-EventLog**命令产生的所有结果都被序列化,之后发送到本地计算机,最后在本机反序列化为对象,再通过管道传递给**Where**做筛选。相对而言,第二个版本的命令效率更为低下,因为会有大量不必要的数据通过网络传输,然后在本地计算机上筛选来自3台计算机的返回结果,而并不是在3台计算机上筛选好结果之后再发送给本地计算机。所以采用第二个版本的命令是一个非常糟糕的主意。

让我们看看其他命令的两个版本,如下面的命令:

```
Invoke-Command -ComputerName Server-R2  
➔-Command {Get-Process -Name Notepad}|  
➔Stop-Process
```

然后看另外一个版本:

```
Invoke-Command -ComputerName Server-R2  
➔-Command {Get-Process -Name Notepad |  
➔Stop-Process}
```

和上面一样,这里唯一的差异是一个大括号的位置不同。但是在本示例中,第一个版本的命令根本无法执行。

仔细对比：我们将**Get-Process-Name Notepad**命令发送到远程计算机。远程计算机获取特定的进程，然后将其序列化到XML，最后通过网络传递给本机。本地计算机收到该XML后，反序列为一个对象，然后通过管道传递给**Stop-Process**。此时问题出现了，本地计算机上被反序列的XML文件中并没有信息表明该进程来自于远程计算机。相反，本地计算机尝试关闭本地运行的**NotePad**进程，但是这根本就不是我们所期望的结果。

这个故事告诉我们，我们需要在远程计算机上完成尽量多的工作。我们唯一需要注意的是如何处理**Invoke-Command**命令的结果，要么显示，要么将结果存储为一个报表或者一个数据文件等。第二个版的脚本正是采用了该思想：发送给远程计算机的命令是 **Get-Process-Name Notepad | Stop-Process**，所以整个命令（包含获取进程以及停止进程）都是在远程计算机上执行。因为**Stop-Process**命令不会产生任何执行结果，并没有任何对象需要序列化传递给我们，所以在我们本地的控制台中无法看到任何信息。但是该命令正好达到我们的目的：停止远程计算机的**NotePad**进程，而不是停止本地计算机上的**Notepad**。

当我们使用**Invoke-Command**命令时，我们总会看到后面跟着一些命令。如果这些命令是用作格式化或者导出数据，那没问题，因为PowerShell可以这样处理**Invoke-Command**的输出结果。但是如果**Invoke-Command**后面跟着操作类型的**Cmdlet**（比如开启、停止、设置或者修改等其他操作），我们需要好好想想我们在做什么。理想情况下，我们希望所有的操作都是运行在远程计算机，而不是本地计算机上。

13.5.3 反序列化对象

远程处理的另一个需要注意的事项是返回给本地计算机的对象可能会缺失部分功能。在大部分情况中，由于它们不再需要关联到可用软件，它们都会缺少对应的方法（**Method**）。

比如，在你本地计算机上运行下面的命令，你会发现存在关联到**Service-Controller**对象的多个方法。

```
PS C:\>Get-Service | Get-Member

        TypeName: System.ServiceProcess.ServiceController
Name      MemberType Definition
-----
Name      AliasProperty Name = ServiceName
RequiredServices AliasProperty RequiredServices = ServicesDep
Disposed  Event        System.EventHandler Disposed(S
Close     Method        System.Void Close()
```

Continue	Method	System.Void Continue()
CreateObjRef	Method	System.Runtime.Remoting.ObjRef
Dispose	Method	System.Void Dispose()
Equals	Method	bool Equals(System.Object obj)
ExecuteCommand	Method	System.Void ExecuteCommand(int
GetHashCode	Method	int GetHashCode()
GetLifetimeService	Method	System.Object GetLifetimeServi
GetType	Method	type GetType()
InitializeLifetimeService	Method	System.Object InitializeLifeti
Pause	Method	System.Void Pause()
Refresh	Method	System.Void Refresh()
Start	Method	System.Void Start(), System.Vo
Stop	Method	System.Void Stop()
WaitForStatus	Method	System.Void WaitForStatus(Syst
CanPauseAndContinue	Property	bool CanPauseAndContinue {get;
CanShutdown	Property	bool CanShutdown {get;}
CanStop	Property	bool CanStop {get;}
Container	Property	System.ComponentModel.IContain
DependentServices	Property	System.ServiceProcess.ServiceC

现在通过远程处理获取类似的对象:

```
PS C:\> Invoke-Command -ScriptBlock { Get-Service } -ComputerName
WCMIS034 | Get-Member

    TypeName: Deserialized.System.ServiceProcess.ServiceController
Name      MemberType      Definition
----      -
ToString  Method          string ToString(), string ToString(string
    format, System.I
Name      NoteProperty    System.String Name=AeLookupSvc
PSComputerName NoteProperty    System.String
PSComputerName=WCMIS034
PSShowComputerName NoteProperty    System.Boolean
PSShowComputerName=True
RequiredServices NoteProperty
    Deserialized.System.ServiceProcess.ServiceController[] Req
RunspaceId NoteProperty    System.Guid RunspaceId=6dc9e130-f7b2-
4db4-
    8b0d-3863033d7df
CanPauseAndContinue Property      System.Boolean {get;set;}
CanShutdown    Property      System.Boolean {get;set;}
CanStop        Property      System.Boolean {get;set;}
Container      Property      {get;set;}
DependentServices Property
    Deserialized.System.ServiceProcess.ServiceController[] {ge
DisplayName    Property      System.String {get;set;}
MachineName    Property      System.String {get;set;}
ServiceHandle  Property      System.String {get;set;}
ServiceName    Property      System.String {get;set;}
ServicesDependedOn Property
```

Deserialized.System.ServiceProcess.ServiceController[] {ge		
ServiceType	Property	System.String {get;set;}
Site	Property	{get;set;}
Status	Property	System.String {get;set;}

查看上面返回的结果，你会发现，除了每个对象都拥有的普通 `ToString()` 方法外，其他的方法都不在了。返回的结果只是对象的一些副本，你无法让它完成停止、暂停、恢复等等操作。所以如果希望对返回的结果执行一些操作，那么这部分命令都应该包含在发送给远程计算机的脚本中；只有这样，这些对象才是可用的，并且会仍然保留它们包含的方法。

13.6 深入探讨

前面的示例都是采用即席远程连接，也就是说，每次都需要我们指定计算机名称。如果你需要在很短时间内多次重复连接到相同的远程计算机，那么你可以创建可重用的持久性连接。我们会在第20章中讲解该技术。

当然，我们也承认并不是每家公司都允许开启 **PowerShell** 的远程处理机制，至少当前不是。比如，那些拥有非常严格安全策略的公司会在所有的客户端和服务端计算机上都会开启防火墙，这将阻止 **PowerShell** 远程处理的连接。如果你所在的公司也是这样，那么你需要确认一下在防火墙中是否有针对远程桌面协议（**RDP**）设置一个例外。我们可以发现，在大部分公司总是会存在该例外，因为管理员总是需要不定时远程连接到某些服务器。如果 **RDP** 是允许使用的，那么也请尝试对 **PowerShell** 的远程处理设置类似例外。因为远程处理连接可以被审核到（它们类似于网络账号，就像访问一个共享文件会在审计日志中出现对应日志），所以它们默认情况下被限制为仅管理员可以连接。在安全风险方面，**PowerShell** 的远程处理和 **RDP** 没多大差别，并且相对于 **RDP**，**PowerShell** 的远程处理在远程计算机上会占用更少的开销。

13.7 远程处理的配置选项

通过阅读帮助文档，可以看到 `Invoke-Command` 和 `Enter-PSSession` 命令都有一个 `-SessionOption` 参数（该参数能处理 `PSSessionOption` 类型对象）。该参数有什么功能呢？

正如我们刚才解释的，这两个命令在运行时都会初始化一个新的PowerShell连接或者会话。它们完成对应的工作后，会自动关闭该会话。一个会话选项（Session Option）是指你可以用来改变建立会话方式的一组选项。我们使用New-PSSessionOption命令来实现该功能。你可以使用该命令实现下面的功能：

- 打开，取消和空闲超时；
- 取消正常数据流的压缩或者加密功能；
- 通过代理服务器传递网络流量时，也可以设置一些代理相关的选项；
- 忽略远程机器的SSL证书、名称以及其他安全特性。

比如，通过下面的命令可以忽略机器名称的检查来打开一个会话。

```
PS C:\> Enter-PSSession -ComputerName Wcmis034
➔-SessionOption (New-PSSessionOption -SkipCNCheck)
[WCMIS034] : PS C:\Users\wh42\Documents>
```

重新查看New-PSSessionOption命令的帮助信息，确认该命令可实现的功能；在第20章中，我们会使用一些选项来完成某些进阶的远程处理任务。

13.8 常见误区

我们在教学课程中讲解远程处理时，总会看到一些常见的问题：

- 默认情况下，只有指定远程计算机的真实名称时，远程处理才能正常工作。不能使用DNS的别名或者IP地址。在第23章中，我们会讨论该限制的背景，同时会给出解决该问题的方案。
- 设计远程处理功能的目的主要是解决域中自动化配置的事情。如果涉及的计算机以及所使用的用户账号都属于同一个域或者可信任的域中，那么一切都会很轻易地实现。如果不是这种场景，那么需要详细查看AboutRemote TroubleShooting的帮助文档。一个需要确认的情形是你是否跨域进行远程处理。如果确认如此，那么你必须修改一些配置选项使得PowerShell可以正常执行，帮助文档中详细描述了该场景。
- 当调用一个命令时，远程计算机会发起一个PowerShell会话。运行你键入的命令，之后关闭PowerShell会话。当你在相同计算机上执行下一条命令时，又会重复该步骤（第一次调用过程中运行的任何结果或者命令都不再有效）。如果你需要运行一系列关联的命令，那么你需要将它们放入相同的调用进程中。

- 确保你以管理员身份去运行PowerShell，特别是对于开启用户账户控制（UAC）功能的计算机。如果你使用的账号在远程计算机上没有管理员权限，那么你需要使用Enter-PSSession或者Invoke-Command命令的-Credential参数去指定另外一个拥有管理员权限的账号。
- 如果你使用的不是Windows防火墙，而是一种第三方防火墙产品，Enable-PSRemoting不会建立特定的防火墙例外。那么你需要手动来完成该项设置。如果远程连接需要穿过一个部署在路由器或者代理服务器上的普通防火墙，那么也需要针对远程流量手动设置一个例外。
- 请不要忘记一点规则，在组策略对象（GPO）中的配置选项会覆盖本地配置的选项。我们经常会看到管理员会花费几小时来使得远程处理可以正常工作，最后才发现一个GPO对象覆盖他们设置的选项。在某些情况下，可能一些好心的同事很久之前设置了一些GPO对象，但是后来忘记了。所以请不要想当然以为没有GPO影响到你的设置，你需要去检查一下，以便确认。

13.9 动手实验

注意： 在该动手实验环境中，你仍然需要运行PowerShell v3或者之后版本的计算机。理论上，你需要在同一活动目录中的两台计算机。但是如果只有一台计算机可以用来进行实验，那么也没关系。

现在，我们需要把本章学到的关于远程处理的知识和前面章节学习到的内容关联起来。你可以尝试是否能完成下面的任务。

1. 创建针对一台远程计算机一对一的连接（如果只有一台计算机，请使用localhost模拟）。打开NotePad.exe，发生了什么？

2. 使用Invoke-Command命令获取一台或者两台远程计算机上尚未开启的服务列表（如果仅有一台计算机，也可以两次使用localhost模拟）。将结果集格式化为一个较宽的列表（提示：也可以在远程计算机上获取结果之后，在本地计算机格式化结果集——也就是说，不要将Format- Cmdlet放入到发送给远程计算机的命令中）。

3. 按照虚拟内存使用排列，使用Invoke-Command命令去获取消耗虚拟内存最高的10个进程。如果可以的话，在一台或者两台远程计算机上运行该命令；如果只有一台计算机，那么在localhost上运行两次。

4. 创建一个文本文件，其中每一行代表一个计算机名称，总共三行数据。如果你只能访问到本地计算机，那么每一行可以是相同的计算机名称

或者localhost。然后在文本文件中列出的计算机上执行Invoke-Command命令去获取最新的100条应用程序的事件日志。

13.10 进一步学习

其实在该书中，我们本可以讲解更多关于PowerShell远程处理的知识，但是这样需要你花费更长的时间（一个月甚至更多）来完成阅读学习。但不幸的是，一些比较棘手的问题都没有得到很好的记录。我们建议你访问网站<http://PowerShellBooks.com>。在该网站上，Don和MVP Dr. Tobias Weltner制作了比较全面地（也是完全免费）探究PowerShell远程处理原理的一本迷你电子书。该电子书中会重讲本章中所学的一些基础知识，但是内容主要集中在比较详细的关于如何配置各种远程处理场景的Step-by-step说明（同时配有彩色截图）。同时，该电子书也会定期更新，所以你需要每隔几个月就检查一遍，以便确认获取的版本为当前最新的版本。当然，你也别忘了，可以通过网站<http://bit.ly/AskDon>向Don咨询一些问题。

第14章 Windows管理规范

我们一直期望但又害怕写这一章。Windows管理规范（Windows Management Instrumentation, WMI）可能是微软提供给管理员使用的最优秀的工具之一。但同时它也是这个公司曾经对我们造成过的最多问题的部分。WMI可以从计算机中收集大量系统信息。但有时候这些信息不易看懂，另外文档也不够友好。在本章，我们将从PowerShell的角度介绍WMI，以及WMI的工作方式和一些不完美的地方，以便全面揭示你将会遇到的问题。

需要强调的是，WMI是一个外部技术；PowerShell仅仅与其接口交互而已。本章的重点将放在PowerShell如何与WMI交互，而不是WMI的底层实现机制。如果你不想深入探讨WMI，我们在本章结尾提供了一些建议。PowerShell已经在最大程度减少你需要与WMI交互的部分做出了改进。

14.1 WMI概要

典型的Windows计算机包含数万个管理信息，WMI会把这些收集并整理成一些尽可能通俗易懂的信息。

在最顶层，WMI被组织成命名空间（namespaces）。可以把命名空间想象为关联到特定产品或技术的一个文件夹。比如，“root\CIMv2”这个命名空间包含了所有Windows操作系统和计算机硬件信息。而“root\MicrosoftDNS”命名空间包含了所有关于DNS服务器（假设你已经把这个角色安装到计算机上）的信息。在客户端计算机上，“root\SecurityCenter”包含了关于防火墙、杀毒软件和反流氓软件这些工具的信息。

注意：“root\SecurityCenter”的内容根据你计算机上的已安装程序的情况而有所不同，新版本的Windows使用“root\SecurityCenter2”代替，这是其中一个WMI使人困惑的地方。

图14.1展示了通过微软管理控制台（Microsoft Management Console, MMC）的WMI控制单元在我本机上产生的一些命名空间。

在命名空间中，WMI被分成一系列的类，每个类是可用于WMI查询的管理单元。比如，在“root\SecurityCenter”中的“Antivirus-Product”类被设计用于保存反间谍软件的信息；在“root\CIMv2”中的“Win32_LogicalDisk”类被设计用于保存逻辑磁盘的信息。但是即使一个计算机上存在某个类，也不代

表计算机实际上安装了这些对应的组件。比如“Win32_TapeDrive”类在所有的Windows版本上都存在，不管是否安装了磁带驱动程序。



图14.1 浏览WMI命名空间

注意： 再一次提醒，不是所有的计算机都包含相同的WMI命名空间或类。比如，新版本的Windows存在“Root\SecurityCenter2”命名空间，而不是“Root\SecurityCenter”命名空间；而前者在新版本的计算机中包含了所有信息。

下面看一下从“root\SecurityCenter2”中查询“AntiSpywareProduct”，你可以查看返回结果：

```
PS C:\> Get-CimInstance -Namespace root\securitycenter2 -ClassName  
antispyw  
areproduct
```

注意： 此例要求PowerShell v3及以上版本，我们稍微介绍一下“Get-CimInstance”命令。

当你有一个或多个可管理组件时，你可以看到在对应的类中有相同数量的实例（instances）。一个实例代表了一个现实世界的事件。如果你的计算机只有一个单一的BIOS，那么在“root\CIMv2”中会有一个关于“Win32_BIOS”的实例。如果计算机安装了100个后台服务，你会看到100个“Win32_Service”的实例。请注意，在“root\CIMv2”中的类型一般以“Wim32_”（即使在64位系统中亦然）或“CIM_”（Common Information Model的缩写，是WMI建立的标准）开头。在其他命名空间中，这些类名前缀很少出现。还有一种情况是在多个命名空间中存在重复的类型，虽然很少。但在WMI中允许，因为每个命名空间实际上是一种有边界的容器。当你引用一个类时，你同时需要引用其命名空间，以便WMI知道从哪里找到对应的类，从而避免因为多个重名但不属于同一个命名空间的类造成混乱。

所有这些实例、类和其他不可名状的东西统称为WMI仓库（WMI repository）。在旧版本的Windows中，WMI仓库有时会损坏从而不可用，必须通过重建来恢复。但是从Win 7开始，这种情况越来越少见。

表面看上去，使用WMI十分简单：你只需要指出哪个类包含你要的信息，然后从WMI中查询类的实例，最后检查实例的属性得知其管理信息。有时候可能需要实例执行一个方法，从而启动一个动作（action）或开始一个配置变更（configuration change）。

14.2 关于WMI的坏消息

WMI在其大部分生命周期中（最近有所好转），微软都没有把过多的精力放在对其内部控制上。微软为WMI制定了一系列的编程标准，但是产品组或多或少把精力放在如何实现类和是否对其文档化。结果就是使得WMI变得混乱。

在“root\CIMv2”命名空间中，有些类提供了让你修改配置设置的方法（methods）。因为属性是只读的，意味着你必须使用方法来修改。如果对应的方法不存在，你就不能使用WMI来修改这些类。当IIS团队采用WMI（IIS第六版），它们针对大量的元素进行了并行类的实现。比如一个网站，可以用一个具有典型只读属性的类表示，但是同时也提供了第二个类用于修改属性。这种情况下很容易因为文档质量不佳而导致混乱，特别是IIS团队倾向于使用自身提供的工具。IIS团队已经放弃了把WMI作为管理接口的做法，并从v7.5开始把注意力集中在PowerShell Cmdlets上，并且用一个PSProvider类替代WMI。

微软从来没规定某个产品必须使用WMI，或者如果这个产品使用了WMI，必须公开WMI的每个可能的部分。微软的DHCP服务可以访问到WMI。正如旧版本的Windows服务器一样，你可以查询网卡的配置，但是不能查到连接速度，因为这些信息不支持通过WMI查询。同时，虽然大部分“Win32_”的类都有很好的文档支持，但其他命名空间下的类大部分都没有相关文档，WMI是不支持类搜索的，因此查找这些类对你来说就变得费时费力（在下节将告诉你如何减少这种影响）。

但是微软也对此进行了改善，微软正在努力使PowerShell Cmdlets尽可能完成更多的管理任务。比如，过去WMI仅用于某种特定的编程方式来重启远程计算机，这个方法由“Win32_OperatingSystem”类实现。而现在，PowerShell提供了名称为“Restart-Computer”的Cmdlet来实现。在某些情况下，Cmdlets内部会通过WMI实现，而无须直接调用WMI。Cmdlets能提供更一致的接口，并且这些接口大部分都有很好的文档支持。虽然WMI不会消失，但时不时，你还是需要在某些场景用到WMI。

实际上，在PowerShell v3及后续版本中，你会留意到大量“CIM”命令，如图14.2所示（作为“Get-Command”输出的一部分）。在大部分情况下，这些命令都是对WMI的某些部分进行了封装，从而提供了以PowerShell为中心与WMI交互的方式。你可以像使用其他Cmdlet一样使用这些Cmdlet，包括对这些Cmdlet使用Help命令。这使使用这部分Cmdlet和使用其他PowerShell中的Cmdlet的体验变得一致，也便于隐藏一些底层的WMI的复杂性。

14.3 探索WMI

最佳的WMI入门恐怕是暂时抛开PowerShell并从WMI自身出发。这里我们使用来自于SAPIEN Technologies公司

（<http://www.sapien.com/downloads>，要求注册的免费软件）提供WMI探索工具来进行WMI研究。我们可以从这个工具中得到大部分所需的关于WMI的信息。当然，这个工作要求耐心和不少的浏览量——这并不是最佳方法，但是我们最终还是选择了这个工具。

首先你不需要安装工具。也就是说，它是绿色的，可以通过U盘等工具把软件保存到任何你需要的计算机上使用。由于每台计算机上的WMI命名空间和类都不尽相同，所以你需要把工具直接在准备查询的机器上运行，以便看到对应机器的WMI仓库。

```

管理员: Windows PowerShell
PS E:\WINDOWS\system32> gcm

CommandType      Name                                     ModuleName
-----
Alias            Add-ProvisionedAppxPackage            Dism
Alias            Apply-WindowsUnattend                 Dism
Alias            Begin-WebCommitDelay                  WebAdminis
Alias            End-WebCommitDelay                    WebAdminis
Alias            Flush-Volume                          Storage
Alias            Get-PhysicalDiskSNV                   Storage
Alias            Get-ProvisionedAppxPackage            Dism
Alias            Initialize-Volume                     Storage
Alias            Move-SmbClient                        Smbwitness
Alias            Remove-ProvisionedAppxPackage          Dism
Alias            Write-FileSystemCache                 Storage
Function         A:
Function         Add-BCDataCacheExtension               BranchCach
Function         Add-BitLockerKeyProtector              BitLocker
Function         Add-DnsClientNrptRule                  DnsClient
Function         Add-DtcClusterTMMapping                MsDtc
Function         Add-InitiatorIdToMaskingSet            Storage
Function         Add-MpPreference                       Defender
Function         Add-NetEventNetworkAdapter             NetEventPa
Function         Add-NetEventPacketCaptureProvider       NetEventPa
Function         Add-NetEventProvider                   NetEventPa
Function         Add-NetEventVmNetworkAdapter           NetEventPa
Function         Add-NetEventVmSwitch                   NetEventPa
Function         Add-NetIPHttpsCertBinding              NetworkTra
Function         Add-NetLbfoTeamMember                  NetLbfo
Function         Add-NetLbfoTeamNic                      NetLbfo
Function         Add-NetNatExternalAddress              NetNat
Function         Add-NetNatStaticMapping                 NetNat
Function         Add-NetSwitchTeamMember                 NetSwitchT
Function         Add-OdbcDsn                            wdac
Function         Add-PartitionAccessPath                 Storage
Function         Add-PhysicalDisk                       Storage
Function         Add-Printer                             PrintManag

```

图14.2 “CIM”命令在WMI类的包装

现在我们需要查询一组计算机并从中得知它们的图标间距设置。这个任务依赖于Windows桌面，并且是操作系统的核心部分，所以我们从“root\CIMv2”类开始，显示在WMI浏览器左侧的树型视图中（见图14.3）。单击命名空间并在右侧查看对应的类，我们知道需要“Desktop”这个关键字。滚动右侧窗口的滚动条，最终锁定“Win32_Desktop”并单击它。此时下方窗体将展示其对应明细，然后我们选择【属性】标签页并查看其内容。在大约三分之一的地方，找到“IconSpacing”，其值为整数。

显而易见，搜索引擎是查询所需类信息的另一种好方法。我们往往使用“WMI”作为“WMI图标间距”的前缀查询，一般在查看几个例子之后就可以找到大概位置。这些例子可能是与VBScript相关的，或者是类似C#或Visual Basic等.NET语言相关的。不过这不重要，因为我们只是在查找WMI类名称。比如，我们在搜索引擎（例子使用Google）中查找“wmi icon spacing”，可能会首先显示<http://stackoverflow.com/questions/202971/formula-or-api-for-calulating-desktop-icon-spacing-on-windows-xp> 作为第一个结果。在这个网页中会有一些C#代码：

```
ManagementObjectSearcher searcher = new
    ManagementObjectSearcher("root\\CIMV2", "SELECT * FROM
Win32_Desktop");
```

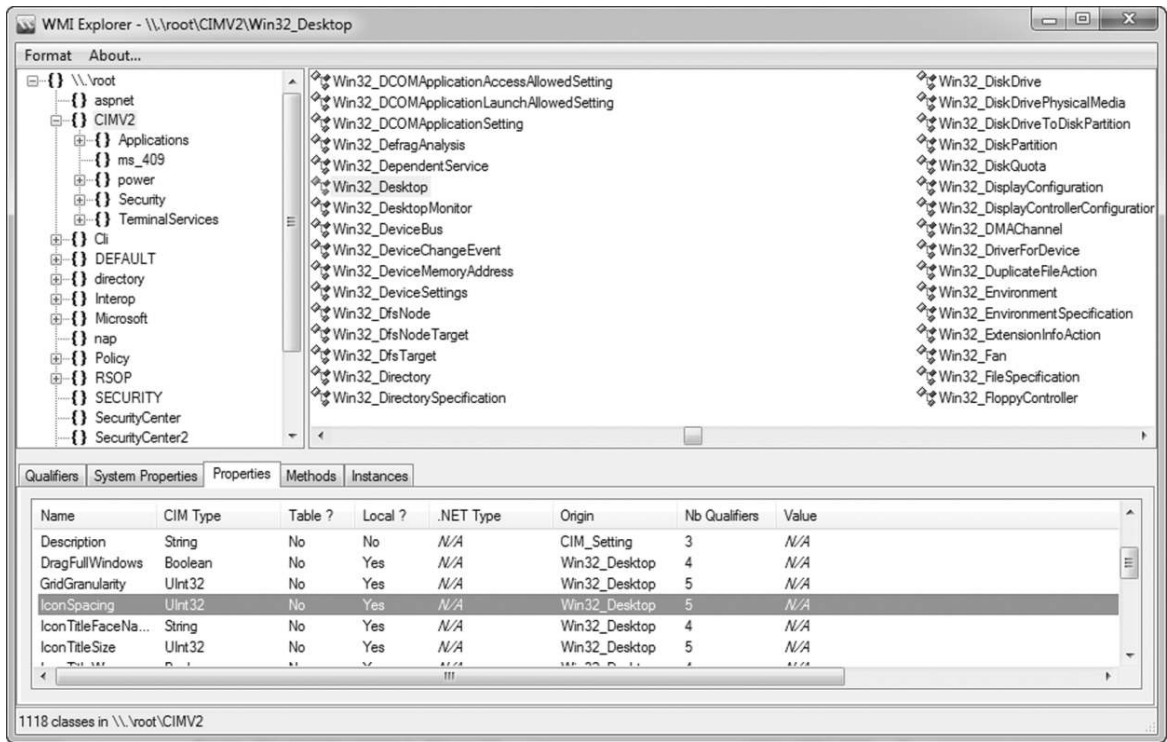


图14.3 WMI浏览器

对此，我们并不知道是否有用，但是“Win32_Desktop”看上去像是个类名。接下来我们就要查询该类名，但这种查询不会在意是否存在相应文档。我们会在本章后续介绍一些关于文档的问题。

另外一个途径是使用PowerShell本身。比如，假设我们想知道一些关于磁盘的信息，那么需要从猜测合理的命名空间开始。但是我们已经知道“root\CIMv2”包含了所有OS核心和硬件设备的信息，所以可以使用下面的命令：

```
PS C:\> Get-WmiObject -Namespace root\CIMv2 -list | where name -like '*dis*'
```

```

    NameSpace: ROOT\CIMv2
Name          Methods          Properties
-----
Win32_DisplayConfiguration {}          {BitsPerPel,
```

Caption, ...}		
Win32_DisplayControllerConfigura...	{}	{BitsPerPixel,
Caption, ...}		
CIM_DiskSpaceCheck	{Invoke}	...}
CIM_DiscreteSensor	{SetPowerState, R...	
{AcceptableValues, Availability, ...		
CIM_Display	{SetPowerState, R...	{Availability,
Caption, ...		
CIM_DiskDrive	{SetPowerState, R...	
{Availability, Capabilities, ...		
Win32_DiskDrive	{SetPowerState, R...	
{Availability, BytesPerSector, ...		
CIM_DisketteDrive	{SetPowerState, R...	
{Availability, Capabilities, ...		
CIM_LogicalDisk	{SetPowerState, R...	{Access,
Availability, BlockSize, ...}		
Win32_LogicalDisk	{SetPowerState, R...	{Access,
Availability, BlockSize, ...}		
Win32_MappedLogicalDisk	{SetPowerState, R...	{Access,
Availability, BlockSize, ...}		
CIM_DiskPartition	{SetPowerState, R...	
{Access, Availability, BlockSize, ...}		
Win32_DiskPartition	{SetPowerState, R...	{Access,
Availability, BlockSize, ...}		
Win32_LogicalDiskRootDirectory	{}	{GroupComponent,
PartComponent}...		
Win32_DiskQuota	{}	{DiskSpaceUsed, Limit, ...}
Win32_LogonSessionMappedDisk	{}	{Antecedent,
Dependent}...		
CIM_LogicalDiskBasedOnPartition	{}	{Antecedent,
Dependent, ...		
Win32_LogicalDiskToPartition	{}	{Antecedent,
Dependent, ...		
CIM_LogicalDiskBasedOnVolumeSet	{}	{Antecedent,
Dependent, ...		
Win32_DiskDrivePhysicalMedia	{}	{Antecedent,
Dependent}...		
CIM_RealizesDiskPartition	{}	{Antecedent,
Dependent, ...		
Win32_DiskDriveToDiskPartition	{}	{Antecedent, Dependent}
Win32_OfflineFilesDiskSpaceLimit	{}	{AutoCacheSizeInMB, ...
Win32_PerfFormattedData_Counters...	{}	{Caption,
Description, ...		
Win32_PerfRawData_Counters_FileS...	{}	{Caption,
Description, ...		
Win32_PerfFormattedData_Distribu...	{}	
{AckMessagesReceivedPersecond, ...		
Win32_PerfRawData_DistributedRou...	{}	
{AckMessagesReceivedPersecond, ...		
Win32_PerfFormattedData_MSDTc_Di...	{}	{AbortedTransactions, ...
Win32_PerfRawData_MSDTc_Distribu...	{}	{AbortedTransactions,

Win32_PerfFormattedData_MSSQLSER...	{}	{Caption,
Description, ...		
Win32_PerfRawData_MSSQLSERVER_SQ...	{}	{Caption,
Description, ...		
Win32_PerfFormattedData_PeerDist...	{}	{BITSBytesfromcache, ...
Win32_PerfRawData_PeerDistSvc_Br...	{}	{BITSBytesfromcache, ...
Win32_PerfFormattedData_PerfDisk...	{}	{AvgDiskBytesPerRead, ...
Win32_PerfRawData_PerfDisk_Logi...	{}	{AvgDiskBytesPerRead, ...
Win32_PerfFormattedData_PerfDisk...	{}	{AvgDiskBytesPerRead, ...
Win32_PerfRawData_PerfDisk_Physi...	{}	{AvgDiskBytesPerRead, ...

最终我们找到“Win32_LogicalDisk”。

注意： 这些以“CIM”开头的名字通常是基本类，所以我们不能直接使用。“Win32”版本的类是Windows特有的，并且这种前缀仅用于特定命名空间——其他空间不使用这种前缀命名方式。

14.4 选择你的武器：WMI或CIM

在PowerShell v3及后续版本中，有两种与WMI交互的方式。

- 所谓的“WMI Cmdlets”，如“Get-WmiObject”和“Invoke-WmiMethod”——这些都是遗留命令，意味着它们依旧能工作，但是微软不会对它们进行后续开发投入。它们与远程过程调用（RPC）交互，也就是说，只有在防火墙支持状态审查时才能通过防火墙（实际上很难）。
- 新版的“CIM Cmdlets”，如“Get-CimInstance”和“Invoke-CimMethod”——它们或多或少等价于旧版本的“WMI Cmdlets”，但是它们通过WS-MAN（由Windows远程管理服务实现）交互，替代原有的RPCs。这是微软的主方向，执行“Get-Command-noun CIM*”可以显示很多微软提供的这类命令的功能。

毫无疑问，这些命令的后端同样是WMI，其差异在于如何交互和如何被使用。在没有安装PowerShell的旧版本系统中，或者没有启用Windows远程管理功能的系统中，WMI Cmdlets依旧能工作（这个功能从Windows NT 4.0 SP3开始引入）。对于已经装有PowerShell和启用了Windows远程管理服务的新系统，CIM Cmdlets提供最佳体验——微软也会对其进行持续的功能及性能改进。

14.5 使用Get-WmiObject

通过“Get-WmiObject” Cmdlet，你可以指定一个命名空间、一个类名甚至远程计算机的名称和其他凭据名。如果需要，还可以从指定的计算机中查询该类的所有实例。

如果需要减少类实例的返回结果，甚至可以提供筛选条件来实现。可以使用下面的语法获取一个命名空间中的类列表：

```
Get-WmiObject -namespace root\cimv2 -list
```

注意，命名空间名字使用的是反斜杠，不是斜杠。

也可以通过指定命名空间和类型查询一个类：

```
Get-WmiObject -namespace root\cimv2 -class win32_desktop
```

其中“root\CIMv2”命名空间是Windows XP SP2及后续版本上的系统默认命名空间，所以如果你的类在这个命名空间中，可以不显式指定。同时，“-class”是位置参数，也就是说，如果你把类名放到第一个位置，它依旧能正常工作。

这里有两个例子，其中一个使用Gwmi别名代替完整的Cmdlet名：

```
PS C:\> Get-WmiObject win32_desktop  
PS C:\> gwmi antispywareproduct -namespace root\securitycenter2
```

动手实验：从现在开始，你应该动手运行每个我们展示的命令。对于涉及远程计算机名称的，如果没有另外一台机器可供测试，可以用localhost替代。

对于许多WMI类，PowerShell的默认配置文件已经设定了需要展示的属性。“Win32_OperatingSystem”是一个很好的例子，因为它默认仅在列表中展示了6个属性。请记住，你总能把WMI对象用管道传输到“Gm”或“Format-List *”中，以便查看所有可用的属性。“Gm”总是列出所有可用的方法。请看例子：

```
PS C:\> Get-WmiObject win32_operatingsystem | gm
```

```

    TypeName:
System.Management.ManagementObject#root\cimv2\Win32_Operating
System

Name                                MemberType  Definition
----                                -
Reboot                             Method      System.Managemen...
SetDateTime                        Method      System.Managemen...
Shutdown                           Method      System.Managemen...
Win32Shutdown                      Method      System.Managemen...
Win32ShutdownTracker              Method      System.Managemen...
BootDevice                         Property    System.String Bo...
BuildNumber                       Property    System.String Bu...
BuildType                         Property    System.String Bu...
Caption                           Property    System.String Ca...
CodeSet                           Property    System.String Co...
CountryCode                       Property    System.String Co...
CreationClassName                  Property    System.String Cr...

```

为了节省空间，这里截断了部分输出结果。如果想看完整结果，请自行执行命令。

另外，“-filter”参数允许你通过指定的规则查询特定实例。这个参数有点棘手。这里有个例子，可以看出其最坏情况下的结果：

```

PS C:\> gwmi -class win32_desktop -filter
"name='COMPANY\\Administrator'"

__GENUS                : 2
__CLASS                : Win32_Desktop
__SUPERCLASS           : CIM_Setting
__DYNASTY              : CIM_Setting
__RELPATH              : Win32_Desktop.Name="COMPANY\\Administrator"
__PROPERTY_COUNT       : 21
__DERIVATION           : {CIM_Setting}
__SERVER               : SERVER-R2
__NAMESPACE            : root\cimv2
__PATH                : \\SERVER-
R2\root\cimv2:Win32_Desktop.Name="COMPANY
\\Administrator"
BorderWidth            : 1
Caption                :
CoolSwitch             :
CursorBlinkRate        : 530
Description            :
DragFullWindows        : False
GridGranularity        :
IconSpacing            : 43
IconTitleFaceName      : Tahoma

```

```
IconTitleSize      : 8
IconTitleWrap      : True
Name               : COMPANY\Administrator
Pattern            : 0
ScreenSaverActive   : False
ScreenSaverExecutable :
ScreenSaverSecure   :
ScreenSaverTimeout  :
SettingID          :
Wallpaper           :
WallpaperStretched  : True
WallpaperTiled      : False
```

对于这个命令和输出结果，有些事情是需要注意的：

- 筛选条件通常被双引号包住。
- 筛选比较操作符并不使用PowerShell的常规操作符“-eq”或“-like”。取而代之的是更加传统、更加编程化的操作符，比如=, >, <, <=, >=和<>。可以使用关键字“LIKE”作为操作符，但在匹配值时必须使用“%”作为字符通配符，如“NAME LIKE ‘%administrator%’”。注意，这里不能像PowerShell的其他地方一样使用*作为通配符。
- 字符串匹配是以单引号包住，这也是筛选表达式的最外层的引号是双引号的原因。
- 避免在WMI中使用反斜杠。当你需要使用文本的反斜杠时，你必须使用两个反斜杠替代。
- Gwmi的输出总会包含一个关于系统属性的数量值。PowerShell的默认显示配置通常会隐藏这个值，但是如果你执意列出所有属性或者这个类不属于默认配置范畴，这个数量值还是可以显示的。系统属性名以双下划线开始。这里有两个非常有用的属性：

__SERVER: 包含被查询的实例所在的计算机名。当所查询的WMI信息来自于多台计算机时非常有用，这个属性来自于“PSComputerName”属性。

__PATH: 是实例本身的绝对应用。如果需要的话，可以用来查询实例。

这个Cmdlet不仅可以从远程计算机中查询信息，也可以从多台计算机中检索，使用一些技巧即可产生一个包含计算机名或IP地址的字符串集合。比如：

```
PS C:\> Gwmi Win32_BIOS -comp server-r2,server3,dc4
```

计算机名按顺序连接，如果某一台计算机不可用，这个Cmdlet会产生一个错误，并跳过这台计算机，继续把后续的计算机连接起来。对于不可用的计算机，Cmdlet通常需要等待直到超时发生，意味着Cmdlet可能会暂停30~45秒之后才决定放弃这台计算机，然后产生错误并继续向后连接。

一旦你查询到一个WMI实例的集合后，可以把它们用管道连接到任何“-Object”Cmdlet、“Format-”Cmdlet或“-Out-”、“Export-”或“-ConvertTo-”Cmdlet中。你可以使用下面的例子定制表格显示“Win32_BIOS”类的信息：

```
PS C:\> Gwmi Win32_BIOS | Format-Table SerialNumber, Version -auto
```

在第10章中，我们已经介绍了如何使用“Format-Table”Cmdlet来产生定制列。当你想从一个给定计算机中查询一些WMI类并集成到一个表时，该技术在这里就可以派上用场。此时你可以创建一个关于表的自定义列，并用列的表达式执行一个全新的WMI查询。语法如下，虽然看上去有点头大，但是结果却能让人满意。

```
PS C:\> gwmi -class win32_bios -computer server-r2, localhost | format-  
table  
  @{l='ComputerName';e={$_.__SERVER}}, @{l='BIOSSerial';e=  
{$_ .SerialNumber}},  
@{l='OSBuild';e={gwmi -class win32_operatingsystem -comp $_.__SERVER |  
select  
ct-object -expand BuildNumber}} -autosize  
  
ComputerName BIOSSerial OSBuild  
-----  
SERVER-R2      VMware-56 4d 45 fc 13 92 de c3-93 5c 40 6b 47 bb 5b 86  
7600
```

如果你把前面的语法复制到PowerShell ISE中，可以很容易编译并格式化：

```
gwmi -class win32_bios -computer server-r2, localhost |  
format-table  
  @{l='ComputerName';e={$_.__SERVER}},  
  @{l='BIOSSerial';e={$_.SerialNumber}},  
  @{l='OSBuild';e={  
    gwmi -class win32_operatingsystem -comp $_.__SERVER |  
    select-object -expand BuildNumber}}
```

```
} -autosize
```

工作原理:

- “Get-WmiObject”从两台计算机中查询“Win32_BIOS”信息。
- 结果被管道传输到“Format-Table”。“Format-Table”被要求创建三个定制列:
 - 第一列: 名称为ComputerName, 使用“Win32_BIOS”实例中的“__SERVER”系统属性得出。
 - 第二列: 名称为BIOSSerial, 使用“Win32_BIOS”实例中的“SerialNumber”属性得出。
 - 第三列: 名称为OSBuild。这列执行一个全新的“Get-WmiObject”查询, 从“Win32_BIOS”实例的“__SERVER”属性中查询“Win32_OperatingSystem”类。然后把结果用管道传输到“Select-Object”中, 这些信息来自于“Win32_OperatingSystem”实例的“BuildNumber”属性的内容, 并用于OSBuild列的填充值。

语法有点复杂, 但是提供了满意的结果。并且作为一个很好的例子展示了如何通过一些精心挑选的PowerShell Cmdlet实现你要的结果。

我们已经提醒过, 一些WMI类包含方法。你可以在第16章中看到如何使用这些方法。这些方法相对难懂, 所以独立出一章来介绍。

14.6 使用Get-CimInstance

Get-CimInstance是PowerShell v3引入的新命令, 与“Get-WmiObject”有很多相似的地方, 但是也有几个语法上的差异:

- 你需要使用“-ClassName”代替“-Class” (虽然你只需要输入-Class, 但是如果你只记住了该参数名称的话, 这没有问题)。
- 不存在用于列出命名空间中的所有类的“-List”参数。取而代之的是使用“Get-CimClass”并搭配“-Namespace”参数来获取类列表。
- 没有“-Credential”参数; 如果你需要从远程计算机查询并被要求提供替代凭据, 需要通过“Invoke-Command” (前面章节已介绍) 发送“Get-CimInstance”。比如:

```
PS C:\> Get-CimInstance -ClassName Win32_LogicalDisk
```

DeviceID	DriveType	ProviderName	VolumeName	Size
----------	-----------	--------------	------------	------

FreeSpace			
-----	-----	-----	-----
A:	2		
C:	3		687173...
580806...			
D:	5	HB1_CCPA_X64F...	358370...
0			

如果你需要使用替代凭据查询远程计算机，可以使用类似命令：

```
PS C:\> invoke-command -ScriptBlock { Get-CimInstance -ClassName win32_proc  
ess } -ComputerName WIN8 -Credential DOMAIN\Administrator
```

14.7 WMI文档

前面提到过，搜索引擎通常是查找已有WMI文档的最佳方式。虽然“Win32_”类在微软的MSDN网站上有很好的文档记录，但是搜索引擎依旧是查询正确页面的最容易的方式。你只需要把类名在Google或者Bing上搜索，通常第一条就会导航到<http://msdn.microsoft.com/>。

14.8 常见误区

在过去的10章里面介绍了如何使用内置的PowerShell帮助，所以你可能更倾向于在PowerShell中运行类似“help win32_service”的命令。不幸的是，在这里行不通。操作系统本身不包含任何WMI信息，所以PowerShell的帮助功能不能实现你的期望。你可能希望从网上的其他管理员和程序员分享的经验中而不是从微软信息中得到大部分你要的信息，比如查询“root\SecurityCenter”。不幸的是，你不会在结果中找哪怕一个微软的文档页。

WMI的筛选规则的差异也是其中一个误区。不管任何时候，你需要在所有可用实例中筛选信息时都应该提供过滤条件。但是别忘了，筛选语法是存在差异的。筛选语法是伴随WMI而不是由PowerShell处理，所以你必须使用WMI规定的语法去替代内置的PowerShell操作符。

另外一些在我们的学生中常见的关于WMI的误区是，虽然PowerShell提供了从WMI查询信息的简易途径，但是WMI并不集成在PowerShell中。

WMI是一个外部技术，有自己的规则和工作方式。WMI虽然可以在PowerShell内部使用，但和其他Cmdlets不一样，因为这些Cmdlets完全集成在PowerShell中。所以请注意WMI的这些误区。

14.9 动手实验

注意： 对于本次动手实验来说，你需要运行PowerShell v3或更新版本PowerShell的计算机。

花点时间完成下面的动手任务。使用WMI的难处主要在于如何找到你所需要的类的信息，所以本实验中大部分时间会花在查找正确的类上面。尝试花点儿时间在思考关键字上（我们会给一些提示），并使用WMI explorer快速查找这些类（WMI Explorer把类按字母顺序排列，便于我们验证自己的猜测）。记住，PowerShell的帮助系统并不能帮助你查找WMI类。

1. 使用什么类可以查看一个网卡的当前IP地址？这个类是否有什么方法可供发布DHCP租期？（提示：network是一个不错的关键字。）

2. 创建一个显示计算机名、操作系统版本号、操作系统描述（标题）和BIOS序列号的表。（提示：你已经见过这个技术，但是你必须稍微反过来使用并且首先查询OS类，然后查询BIOS。）

3. 使用WMI查询关于热修复补丁（hotfixes）的列表。（提示：微软通常把这些引用为quick fix engineering。）这个列表的内容是否和“Get-Hotfix”的结果不一致？

4. 列出关于服务的列表，包含它们的当前状态、启动模式和启动账号信息。

5. 能否找到一个类用于显示已安装的软件产品信息？你认为这个列表完整了吗？

动手实验： 当你完成这个实验后，尝试完成附录中的实验回顾2。

14.10 进一步学习

WMI是一个巨大的、复杂的技术，其中一些技术足以编写一整本书。实际上确实有人这样做了：PowerShell and WMI by fellow MVP Richard

Siddaway(Manning, 2012)。这本书使用了大量例子，并讨论了关于PowerShell v3引入的CIM Cmdlets的新功能。如果谁有深入学习WMI的意愿，我强烈建议阅读这本书。

如果你发现WMI实在很难理解，别担心。这是正常反应。但是我们有一些好消息：在PowerShell v3及后续版本中，你可以轻易使用WMI而不用深入理解它。因为微软已经开发了数百个Cmdlets用于封装WMI。这些Cmdlets提供了帮助信息、可发现性、示例和所有其他好用的Cmdlets能提供给的东西，但是它们的内核还是使用WMI。这样能更好地使用WMI的强大功能，又避免处理一些使人困惑的元素。

第15章 多任务后台作业

每个人都会跟你说“多任务”，对吧？为什么PowerShell不能同时处理多个任务来实现“多任务”呢？事实证明，PowerShell完全可以实现该功能，特别是涉及多台目标计算机的长时间运行的任务时。请确保在学习本章之前，你们已对第13章和第14章进行了阅读，因为在本章中会更加深入地使用这些远程处理和WMI的概念。

15.1 利用PowerShell实现多任务同时处理

在你的印象中，你应该会将PowerShell视作一种单线程的应用程序，也就意味着PowerShell一次只能处理单个任务。你键入一条命令，然后按回车键，之后PowerShell就会等待该命令执行结束。除非第一条命令执行结束，否则你无法运行第二条命令。

但是借助于PowerShell的后台作业功能，它可以将一个命令移至另一个独立的后台线程（一个独立的，PowerShell后台进程）。该功能使得命令以后台模式运行，这样你就可以使用PowerShell处理其他任务。但是你必须在执行该命令之前就决定是否这样处理，因为在按回车键之后，无法将一个正在运行的命令移至后台进程。

当命令处于后台模式时，PowerShell会使用一些机制来查看这些进程的状态，获取产生的结果等。

15.2 同步VS异步

首先介绍一些术语。正常情况下，PowerShell会使用_同步模式_执行命令，也就意味着，在按回车键之后，你需要等待命令执行完毕。将一个命令置于后台模式将会使得该命令_异步_运行，也就是说，直到异步执行的命令结束时，你都可以使用PowerShell处理其他任务。

下面是在两种模式中运行命令时的重要差异之处。

- 当在同步模式下运行命令时，你可以响应输入请求；当使用后台模式运行命令时，根本就没有机会看到输入请求——实际上，当

遇到输入请求时，会停止执行该命令。

- 在同步模式中，如果遇到错误，命令会立即返回错误信息；后台执行的命令也会产生错误信息，但是你无法立即查看这些信息。如果需要，你必须通过一些设定来获取这些信息（第22章会讲解如何实现）。
- 在同步模式中，如果忽略了某个命令的必需参数，PowerShell会提示对应的缺失信息；如果是后台执行的命令，无法进行提示，所以命令将会执行失败。
- 在同步模式中，当命令的执行结果开始产生时，就会立即返回；但是当命令处于后台模式时，你必须等待命令执行结束，才能获得缓存的执行结果。

通常情况下，我们会用同步模式执行命令，以便对这些命令进行测试，并使得可以正常工作。仅当它们被全面调试并能按照预期执行后，我们才会使用后台模式。我们只有遵循这些规则来保证命令的成功执行，这样才是将命令置于后台模式的最好的时机。

PowerShell将后台执行的命令称为作业（**_Jobs_**）。你可以通过多种方法来创建作业，同时存在多个命令来管理它们。

补充说明

严格意义上，本章中讨论的作业只是你将来会使用到的其中一种而已。本质上讲，作业只是PowerShell的一个扩展点，也就是说，对他人（不管是微软还是第三方）而言，都有可能创建其他功能（也命名为作业）。但是这些作业与本章描述的作业看起来并不一样，并且工作方式也不一样。实际上，本章末尾将讲到的调度作业（**Scheduled Jobs**）与本章前面提到的常规作业并不一致。当你为实现不同目的而扩展该Shell时，也会遇到很多其他一些作业。我们只是想让你知道这些小细节，并且理解到本章中所学的知识仅适用于PowerShell原生的常规作业。

15.3 创建本地作业

首先讲到的第一个作业类型应该是最简单的：本地作业。这是一个命令几乎完全运行于你的本地计算机（在后面会讲到对应的例外），并且该命令以后台模式运行。

为了创建这种类型的作业，你需要使用**Start-Job**命令。参数**-ScriptBlock**使得你可以指定需要执行的命令（一个或多个）。**PowerShell**会自动使用默认的作业名称（**Job1**，**Job2**等）。当然，你也可以使用**-Name**参数来指定特定的作业名称。如果你需要作业运行在其他凭据下，那么可以使用**-Credential**参数来接受一个域名\用户名（**DOMAIN\UserName**）的凭据，同时该参数也会提示你输入密码。如果没有指定一个脚本块，你也可以使用**-FilePath**参数来使得作业执行包含多个命令的完整脚本文件。

下面是一个简单的示例。

```
PS C:\> Start-Job -ScriptBlock {Dir}
```

Id	Name	PSJobTypeName	State	HasMoreData	Location
1	Job1	BackgroundJob	Running	True	localhost

该命令的执行结果为新建了一个作业对象，并且正如示例所示，该作业会立即开始运行。同时，该作业会按照顺序被赋予一个作业ID号，正如上面表格所示。

我们认为，这些作业完全运行于本地计算机上，的确如此。如果你执行一个可支持**-ComputerName**参数的命令，在这种情形下，作业中的命令会被允许访问远程计算机。比如下面的示例：

```
PS C:\> Start-Job -ScriptBlock {  
➔Get-EventLogSecurity -Computer Server-R2  
}
```

Id	Name	PSJobTypeName	State	HasMoreData	Location
3	Job3	BackgroundJob	Running	True	localhost

动手实验： 我们期望你能持续跟随并执行所有的命令。如果你仅有一台计算机可以使用，请使用真实的本地计算机名称，同时用**localhost**作为第二台计算机，这样**PowerShell**会采用类似处理两台计算机的方式来执行命令。

作业的进程会在你本地计算机上运行，它会与指定的远程计算机进行连接（比如本示例中的**Server-R2**）。所以从某种程度上说，这个作业就是一个“远程作业”。但是由于该命令实际上是在本地运行，所以我们仍然将它视为本地作业。

细心的读者可能已经注意到，创建的第一个作业被命名为**Job1**，同时ID为1，但是创建的第二个Job名为**Job3**，同时ID为3。原因是，每个作业至少都有包含一个子作业，第一个子作业（**Job1**的子作业）会被命名为**Job2**，其ID为2。在本章后面章节会讲到子作业相关的知识。

另外，需要记住几点：尽管本地作业是在本地运行，但是它们也会需要使用PowerShell的远程处理系统的架构，也就是在第13章中所讲的知识。如果你还没启用远程处理，那么将无法创建本地作业。

15.4 WMI作业

创建作业的另一种方法是使用**Get-WMIObject**命令。正如我们在上一章节所讲，**Get-WMIObject**命令会与一台或多台远程计算机进行连接，但是通过串行方式实现。这意味着如果给出一长串计算机名称，将需要花费很长的时间去执行某命令，那么将该命令移至后台作业就成为了必然选择。为了将该命令置为后台运行模式，像往常一样执行**Get-WMIObject**命令，但是需要加上**-AsJob**参数。此时，你不能指定一个自定义的作业名称，只能使用PowerShell指定的默认作业名称。

动手实验：如果你在测试环境中运行相同的命令，那么需要在C:根目录下新建一个名为**allservers.txt**的文本文件（因为在这些示例中，均在该路径下执行命令），同时按照每行一个名称的格式在该文件中写入多个计算机名称。你可以将本地计算机名称，以及多个**localhost**放在该文件中，正如我们展示的这样。

```
PS C:\> Get-WMIObject Win32_OperatingSystem -ComputerName (
➔Get-Content allservers.txt) -AsJob
```

Id	Name	PSJobTypeName	State	HasMoreData	Location
5	Job5	WmiJob	Running	True	Server-R2,lo...

在该示例中，PowerShell会创建一个上层的父作业（Job5，如上面返回结果中所示），同时会针对指定的每个计算机创建一个子作业。你可以看到，上面的输出表格的Location列中包含多个计算机名称，也就表明该作业也会在这些计算机上运行。

理解到Get-WMIObject命令仅会运行在本地计算机是非常重要的；该命令会使用正常的WMI通信机制与指定的远程计算机进行连接。它仍然一次只在一台计算机上执行，并且遵循直接跳过不可访问的计算机的默认规则等。实际上，该实现过程等同于同步执行Get-WMIObject命令，唯一不同点是该命令在后台运行。

动手实验： 你也会发现存在除Get-WMIObject外的其他命令来启动一个作业。尝试执行Help * -Parameter AsJob，看看你是否可以找到所有的这种命令。

请注意，在第14章中学到的新的Get-CimInstance命令，并没有包含-AsJob参数。如果你想在作业中使用该命令，请运行Start-Job或者Invoke-Command（你即将学到的命令），并且将Get-CimInstance（或者说，任何新的CIM命令）放在脚本块中。

15.5 远程处理作业

下面介绍最后一种可以用来创建新作业的技术：PowerShell的远程处理功能，也就是你在第13章中学习的功能。当使用Get-WMIObject时，你会使用-AsJob参数来实现该功能，但是这里我们会通过将该参数添加到Invoke-Command Cmdlet中来实现。

这里有一个重要的不同点：在-ScriptBlock参数（或者是该参数的别名，-Command）中指定的任意命令都会并行发送到指定的每台计算机。多达32台计算机可以同时被访问（除非你修改了-ThrottleLimit参数来允许同时访问更多或者更少的计算机），所以当你指定了超过32个计算机名称，仅有前32台计算机开始执行该命令。当前32台计算机即将结束时，剩余的计算机才可以开始执行这些命令。另外，当在所有计算机上都执行结束后，上层的父作业会返回一个完整的状态。

不像另外两种新建作业的方式，该技术要求你在每台目标计算机上安装第二版或者之后版本的PowerShell，同时要求在每台目标计算机上PowerShell中均启用远程处理。因为命令会真正运行在每台计算机

上，所以可以通过分布式计算工作负载来提升复杂的或者长时间运行命令的性能。执行结果会返回到你的本地计算机。在你准备查看它们之前，结果都会与作业一起被存储。

在下面的示例中，你可以看到通过 **-JobName** 参数指定一个特有的作业名称，这样就不需要无意义的默认名称。

```
PS C:\> Invoke-Command -Command {Get-Process}
➔-ComputerName (Get-Content .\allservers.txt )
➔-AsJob -JobName MyRemoteJob
```

Id	Name	PSJobTypeName	State	HasMoreData	Location
9	MyRemoteJob	RemoteJob	Running	True	Server-R2, loca...

15.6 获取作业执行结果

当开启一个作业之后，你最想做的第一件事应该就是确认作业是否执行结束。**Get-Job Cmdlet**可以获取在系统中定义的所有作业，并且返回其状态。

```
PS C:\> Get-Job
```

Id	Name	PSJobTypeName	State	HasMoreData	Location
1	Job1	BackgroundJob	Completed	True	localhost
3	Job3	BackgroundJob	Completed	True	localhost
5	Job5	WmiJob	Completed	True	Server-R2, loca...
9	MyRemoteJob	RemoteJob	Completed	True	Server-R2, loca...

你也可以通过作业ID或者名称去查询特定的作业信息。我们建议你尝试该命令并且将返回结果通过管道传递给 **Format-List ***，因为你已经收集了很多有用的信息：

```
PS C:\> get-job -id 1 | format-list *
State      : Completed
HasMoreData : True
```

```
StatusMessage :
Location      : localhost
Command       : dir
JobStateInfo  : Completed
Finished      : System.Threading.ManualResetEvent
InstanceId    : e1ddde9e-81e7-4b18-93c4-4c1d2a5c372c
Id            : 1
Name          : Job1
ChildJobs     : {Job2}
Output        : {}
Error         : {}
Progress      : {}
Verbose       : {}
Debug         : {}
Warning       : {}
```

动手实验：如果你一直跟着执行上面的命令，请记住，你的作业ID以及名称与上面返回的结果不一样。请通过**Get-Job Cmdlet**的结果来获取你环境中的作业ID与名称，然后使用它们来替换上面示例中对应的部分。

其中**ChildJobs**属性是返回信息中最重要的部分之一，在后面会讲到该部分。

为了获取一个作业的执行结果，请使用**Receive-Job**命令。但是在运行该**Cmdlet**之前，请先了解下面的一些知识点。

- 你必须指定希望获取返回结果的对应作业。可以通过作业ID、作业名称，或者通过**Get-Job**命令来获取作业列表，之后将它们通过管道传递给**Receive-Job**命令。
- 如果你获取了父作业的返回结果，那么该结果会包含所有子作业的输出结果。当然，你也可以获取一个或多个子作业的执行结果。
- 正常情况下，当获取了一个作业的返回结果之后，会自动在作业的输出缓存中清除对应的数据，这样你不能再次获取它们。可以通过**-Keep**命令在内存中保留输出结果的一份拷贝。或者如果你希望保存一份拷贝以作它用，也可以将结果输出到**CliXML**中。
- 作业的返回结果可能是反序列化的对象，也就是你在第13章中所学的知识。也就意味着返回的结果是它们产生时的一个快照，它们可能不会包含可以执行的任何方法。但是如果需要的话，你直接将作业的返回结果通过管道传递给一些**Cmdlet**，比如**Sort-**

Object、Format-List、Export-CSV、ConvertTo-HTML、Out-File等。

下面是一个示例。

```
PS C:\>Receive-Job -ID 1

    Directory: C:\Users\Administrator\Documents
Mode                LastWriteTime         Length Name
----                -
d---- 11/21/2009 11:53 AM             Integration Services Script
Component
d---- 11/21/2009 11:53 AM             Integration Services Script Task
d----  4/23/2010  7:54 AM             SQL Server Management Studio
d----  4/23/2010  7:55 AM             Visual Studio 2005
d---- 11/21/2009 11:50 AM             Visual Studio 2008
```

前面的输出展示了一个比较有趣的结果。这里重申起初创建该作业的命令：

```
PS C:\>Start-Job -ScriptBlock { Dir }
```

尽管当运行该命令时，PowerShell是在C:\路径下，但是在结果中的路径却是C:\Users\Administrator\Documents。正如你所见，本地作业运行时会在不同的上下文中，这可能会导致路径的变更。当使用后台作业时，请永远不要猜测这些文件路径。因此需要使用绝对路径来确保你可以关联到作业命令可能需要的任何文件。如果我们希望后台作业获取C:\下的目录信息，那么我们应该这样执行命令：

```
PS C:\>Start-Job -ScriptBlock { Dir C:\ }
```

当我们获取Job1的结果时，我们并没有指定-Keep参数。如果我们再次获取这部分结果，不会得到任何信息，因为这部分结果已经没有了与作业被缓存了。

```
PS C:\>Receive-Job -ID 1
PS C:\>
```

下面的命令展示了如何强制结果驻留在内存缓存中。

```
PS C:\>Receive-Job -ID 3 -Keep
```

Index	Time	EntryType	Source	InstanceID	Message
6542	Oct 04 11:55	SuccessA...	Microsoft-Windows...		4634
An...					
6541	Oct 04 11:55	SuccessA...	Microsoft-Windows...		4624
An...					
6540	Oct 04 11:55	SuccessA...	Microsoft-Windows...		4672
Sp...					
6539	Oct 04 11:54	SuccessA...	Microsoft-Windows...		4634
An...					

你希望最终会释放缓存作业结果的内存，后面会做对应的说明。但是首先，我们快速看一下如何将作业结果通过管道直接传递给其他 Cmdlet。

```
PS C:\>Receive-Job -Name MyRemoteJob | Sort-Object PSComputerName |
➔Format-Table -GroupByPSComputerName
```

PSComputerName: localhost							
Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	ID	ProcessName
PSComputerName							
195	10	2780	5692	30	0.70	484	lsmd
237	38	40704	36920	547	3.17	1244	Micro...
146	17	3260	7192	60	0.20	3492	msdtc
1318	100	42004	28896	154	15.31	476	lsass

该作业是我们通过Invoke-Command命令创建的。和以前一样，该 Cmdlet会添加PSComputerName属性，这样我们就能追踪哪个对象来自

于哪台计算机。因为我们从上层父作业中获取了结果，它包含了我们指定的所有计算机上的作业，这将允许命令可以按照计算机名称对结果进行排序，然后针对每台计算机创建独立的表组。

Get-Job命令也会告知你还有哪些作业还留有剩余的结果。

```
PS C:\>Get-Job

WARNING: column "Command" does not fit into the display and was
removed.

Id      Name      State      HasMoreData  Location
--      -
1       Job1      Completed  False        localhost
3       Job3      Completed  True         localhost
5       Job5      Completed  True         server-r2,lo...
8       MyRemoteJob Completed  False        server-r2,lo...
```

当某个作业的输出结果没有被缓存时，对应的**HasMoreData**列会为**False**。在**Job1**和**MyRemoteJob**这两个场景中，我们已经获取了这部分结果，并且获取时并未指定**-Keep**参数。

15.7 使用子作业

在前面我们提及，所有的作业都由一个上层父作业以及至少一个子作业组成。我们再次查看该作业：

```
PS C:\>Get-Job -ID 1 | Format-List *

State      : Completed
HasMoreData : True
StatusMessage :
Location    :localhost
Command     :dir
JobStateInfo : Completed
Finished    :System.Threading.ManualResetEvent
InstanceId  : e1ddde9e-81e7-4b18-93c4-4c1d2a5c372c
Id          : 1
Name        : Job1
ChildJobs   : {Job2}
Output      : {}
```

```
Error      : {}
Progress   : {}
Verbose    : {}
Debug      : {}
Warning    : {}
```

动手实验： 不要照搬该部分的脚本，因为你如果自始至终都照搬的话，那么你已经获取ID为1的作业结果（也就是说，此时无法再次获取该结果）。如果你希望执行该脚本，那么请执行 **Start-Job -Script{Get-Service}** 来新建一个作业，然后使用该作业ID来替换我们示例中的ID。

你可以看到，Job1包含了一个子作业Job2。既然你知道了它的名字，那么你就可以直接获取该作业的信息。

```
PS C:\>Get-Job -Name Job2 | Format-List *

State      : Completed
StatusMessage :
HasMoreData : True
Location    :localhost
Runspace    :System.Management.Automation.RemoteRunspace
Command     :dir
JobStateInfo : Completed
Finished    :System.Threading.ManualResetEvent
InstanceId   : a21a91e7-549b-4be6-979d-2a896683313c
Id          : 2
Name        : Job2
ChildJobs    : {}
Output       : {Integration Services Script Component, Integration
                es Script Task, SQL Server Management Studio, Visual Studi
                o 2005...}
Error        : {}
Progress     : {}
Verbose      : {}
Debug        : {}
Warning      : {}
```

有些时候，某个作业会包含多个子作业，它们均会以该格式罗列出来。此时你可能希望采用不同的方式来罗列它们，比如下面这样：

```
PS C:\>Get-Job -ID 1 | Select-Object -Expand ChildJobs
```

WARNING: column "Command" does not fit into the display and was removed.

ID	Name	State	HasMoreData	Location
2	Job2	Completed	True	localhost

该技术会针对ID为1的作业创建一个表格来存放子作业。该表格可以采用任意的长度，只要能将它们罗列出来。

你也可以使用带有作业名称或者ID的Receive-Job命令来获取来自任何独立子作业的结果。

15.8 管理作业的命令

针对作业，也可以使用另外三个命令。对这三个命令中任意一个，你都可以指定作业ID、作业名称，或者先获取作业信息，然后通过管道传递这三个命令：

- **Remove-Job**——该命令会移除某个作业，包括从内存中移除针对该作业缓存的任意输出结果。
- **Stop-Job**——如果某个作业看起来卡住了，你可以通过执行该命令来停止它。但是仍然可以获取截止到该时刻产生的任何结果。
- **Wait-Job**——该命令在下面场景中比较有用：当使用一段脚本开启一个作业，同时希望该脚本在作业运行完毕之后继续执行。该命令会使得PowerShell停止并等待作业的执行，在作业执行结束后，允许PowerShell继续执行。

例如，为了移除已经获取了结果的作业，我们可以使用下面的命令。

```
PS C:\>Get-Job | Where { -Not $_.HasMoreData } | Remove-Job
PS C:\>Get-Job
```

WARNING: column "Command" does not fit into the display and was removed.

Id	Name	State	HasMoreData	Location
3	Job3	Completed	True	localhost
5	Job5	Completed	True	server-r2,lo...

在PowerShell中，作业也可以执行失败，也就意味着在执行过程中发生了某些错误。考虑下面的示例：

```
PS C:\>Invoke-Command -Command { Nothing } -Computer NotOnline -
AsJob -Job
Name ThisWillFail

WARNING: column "Command" does not fit into the display and was
removed.
```

Id	Name	State	HasMoreData	Location
11	ThisWillFail	Failed	False	NotOnline

在这里，我们向根本不存在的计算机发送一条不存在的命令来开启一个作业。当然，该作业立即就会失败，正如返回的State列。此时，我们根本就不需要使用Stop-Job，因为该作业并未运行。但是我们仍然可以获取对应的子作业列表：

```
PS C:\>Get-Job -ID 11 | Format-List *
```

```
State      : Failed
HasMoreData : False
StatusMessage :
Location    :notonline
Command     : nothing
JobStateInfo : Failed
Finished    :System.Threading.ManualResetEvent
InstanceId  : d5f47bf7-53db-458d-8a08-07969305820e
ID          : 11
Name        :ThisWillFail
ChildJobs   : {Job12}
Output      : {}
Error       : {}
Progress    : {}
Verbose     : {}
Debug       : {}
```

```
Warning      : {}
```

此时，我们就可以获取其子作业的信息了：

```
PS C:\>Get-Job -Name Job12

WARNING: column "Command" does not fit into the display and was
removed.
ID      Name      State  HasMoreData  Location
---      -
12      Job12     Failed  False        NotOnline
```

正如你所见，该作业并没有产生任何输出，因此你将不能获取对应的结果。但是该作业的错误信息仍然保留在结果中，你可以使用 **Receive-Job** 命令来获取这部分信息：

```
PS C:\>Receive-Job -Name Job12
Receive-Job: [NotOnline]Connecting to remote server failed with the
following
error message:WinRM cannot process the request. The following error
occured
while using Kerberos authentication:The network psth was not found.
```

完整的错误信息很长，在这里我们做了一些截断来节省一些空间。你可以看到，错误信息中包含产生错误的计算机名称：**[NotOnline]**。当仅有某台计算机无法连接时会发生什么呢？我们看下面的示例：

```
PS C:\>Invoke-Command -Command { Nothing }
➡-Computer NotOnline,Server-R2 -AsJob -JobNameThisWillFail
警告: 列"Command"无法显示, 已被删除。
ID      Name      State  HasMoreData  Location
---      -
13      ThisWillFail  Running  True        NotOnline,Se...
```

稍待片刻，再执行下面的命令：

```
PS C:\>Get-Job
警告: 列“Command”无法显示, 已被删除。
ID      Name      State  HasMoreData  Location
---      -
13      ThisWillFail  Failed  False        NotOnline,Se...
```

可以看到该作业仍然失败，但是让我们检查一下独立的子作业状态：

```
PS C:\>Get-Job -ID 13 | Select -Expand ChildJobs
警告: 列“Command”无法显示, 已被删除。
ID      Name      State  HasMoreData  Location
---      -
14      Job14     Failed  False        NotOnline
15      Job15     Failed  False        Server-R2
```

好吧，它们都失败了。我们都能预感到Job14会失败，并且也知道失败的原因，但是Job15怎么了？

```
PS C:\>Receive-Job -Name Job15
Receive-Job : The term 'nothing' is not recognized as the name of a
Cmdlet, function,
script file, or operable program. Check the spelling of the name,
or if a path was
included, verify that the path is correct and try again.
```

对，这就是原因，我们让它执行了一个根本不存在的命令。正如你所见，每一个子作业都会由于不同的原因执行失败，PowerShell能分别进行追踪。

15.9 调度作业

在v3版本的PowerShell中介绍了针对调度作业的支持——可以在Windows的任务计划程序中使用PowerShell友好的方式创建任务。这里的作业与之前讲的那些作业相比，会采用不同的工作方式。正如前面写到的，作业是PowerShell中的一个扩展点，也就意味着允许存在多种

通过不同方式实现的作业。调度作业正好是这些不同种类的作业中的一种。

你通过创建一个触发器（**New-JobTrigger**）来开启一个调度作业，该触发器主要用作定义了任务的运行时间。同时，你也可以使用**New-ScheduledTaskOption**命令来设置该作业的选项。之后你使用**Register-ScheduledJob**命令将该作业注册到任务计划程序中。该命令采用任务计划程序中的XML格式来创建作业的定义，之后在磁盘上新建一个文件夹的层次结构来存放每次作业运行的结果。

现在看下面的示例：

```
PS C:\> Register-ScheduledJob -Name DailyProcList -ScriptBlock {
Get-Process }
-Trigger (New-JobTrigger -Daily -At 2am) -ScheduledJobOption
(New-ScheduledJobOption -WakeToRun -RunElevated)
```

警告：列“Enabled”无法显示，已被删除。

ID	Name	JobTriggers	Command
1	DailyProcList	{1}	Get-Process

该命令会新建一个作业，该作业在每天凌晨两点执行**Get-Process**命令。如果有必要，会唤醒计算机，同时要求该作业运行在高级特权下。当作业执行完毕后，你可以回到**PowerShell**中，执行**Get-Job**来查看每次该调度作业执行结束时的一个标准作业列表。

```
PS C:\>Get-Job
```

警告：列“Command”无法显示，已被删除。

ID	Name	State	HasMoreData	Location
6	DailyProcList	Completed	True	localhost
9	DailyProcList	Completed	True	localhost

不像常规的作业，从调度作业中获取结果并不会导致结果被删除，因为它们是被缓存在磁盘上，而非内存中。之后可以继续多次获取该结果。当你移除这些作业时，对应的结果也会从磁盘上被移除。

如图15.1所示，输出的结果会存放于磁盘上特定的文件夹中，**Receive-Job**命令可以阅读这些结果。

你可以通过**Register-ScheduledJob**命令的**-MaxResultCount**参数来控制存放结果的数量。

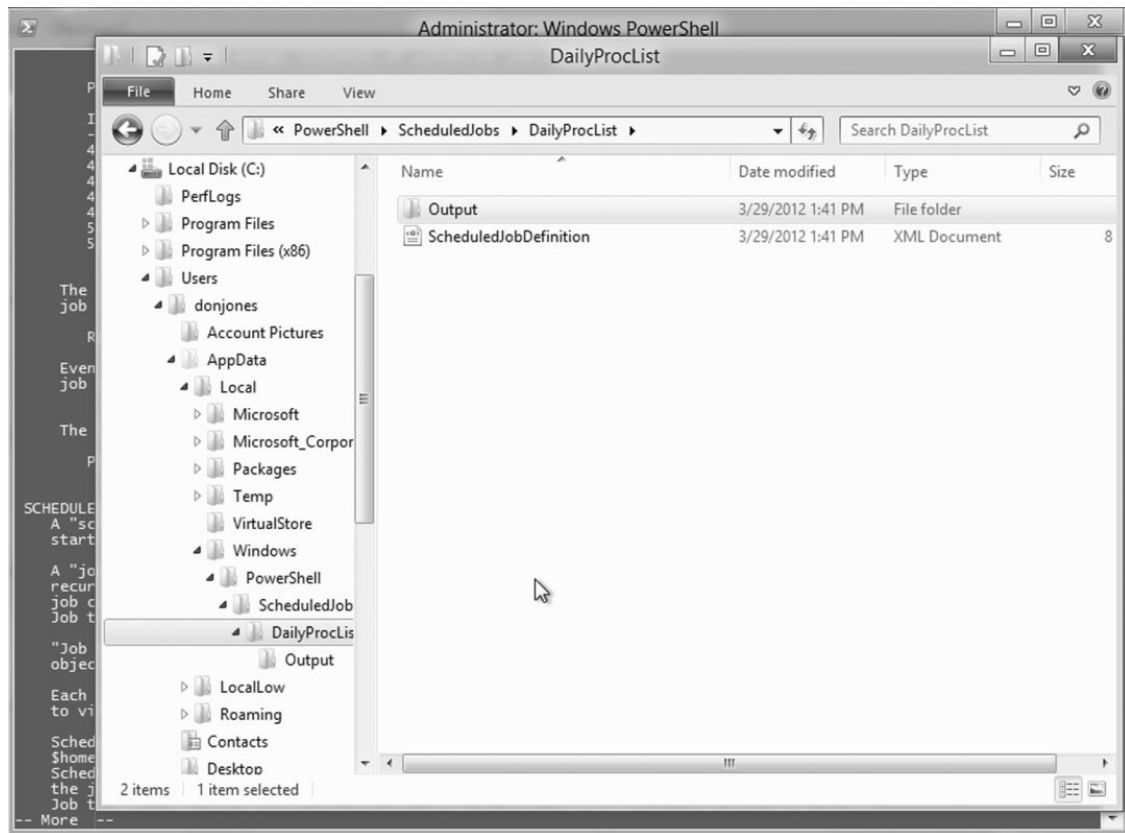


图15.1 调度作业的输出结果存放于磁盘

15.10 常见困惑点

一般情况下，作业都是比较简单的，但是我们曾经见到其他人经常慌乱地完成一件事。请不要这样做：

```
PS C:\>Invoke-Command -Command { Start-Job -ScriptBlock { Dir } }  
➡-ComputerName Server-R2
```

执行该命令之后，会对**Server-R2**计算机开启一个临时的连接，并且在该计算机上开启一个本地作业。遗憾的是，该连接会立即中断，这样就导致你无法重新连接并且获取该作业的信息。一般而言，不要混淆和随意匹配开启作业的三种方式。

比如下面的命令也是一个很差的想法。

```
PS C:\>Start-Job -ScriptBlock { Invoke-Command -Command { Dir }  
➡-ComputerName SERVER-R2 }
```

该命令太冗长了；完全可以通过保留**Invoke-Command**部分，之后使用**-AsJob**参数来使得该作业被后台运行。

更少的困惑，但同样有趣的是教室里学生经常问到的关于作业的一些问题。其中最重要的一个问题可能是“我们是否可以看到由其他人开启的作业呢”，这里的答案是“不能”，但是调度作业例外。常规的作业完全包含在**PowerShell**进程中。尽管你可以看到其他用户在运行**PowerShell**，但是你还是没有办法看到该进程内部的一些信息。这和其他应用程序一样。例如，你可以看到他人有运行微软的**Office Word**软件，但是你无法看到他们正在编辑的文档，因为这些文档完全隐藏于**Word**的进程中。

仅当**PowerShell**进程开启，作业才会维持。当你关闭进程后，在进程中定义的任何作业就会消失。无法在**PowerShell**外部的任意地方定义作业，所以它们依赖于继续运行的进程，保证可以自行维护。

针对前面的论述，调度作业是一个例外：具有权限的任何人都可以看到它们，修改它们，删除它们，以及获取它们的结果。这是因为它们存放于物理磁盘上。请注意，它们存放于你的用户配置文件下，因此它通常要求管理员从配置文件中获取文件（和结果）。

15.11 动手实验

动手实验：对于本章的动手实验环节，你需要操作系统为**Windows 8**（或之后）或者**Windows Server 2012**（或之后）运行**PowerShell**的计算机。

下面的实验应该能帮助你理解如何在PowerShell中使用各种类型的作业以及任务。在进行这些实验时，请不要要求自己仅通过单行代码实现。某些时候，可能将它们拆成独立的步骤会更容易。

1. 创建一次性的后台作业来寻找C:驱动器中所有的PowerShell脚本。需要很长时间运行完成的任务都是一个作业的有效候选者。

2. 你意识到该后台作业有助于在一些服务器上识别所有PowerShell脚本。你如何在一组远程计算机上运行任务1中相同的命令呢？

3. 创建一个后台作业来获取计算机上系统事件日志中最近的25条错误记录，之后将记录导出为CliXML。你期望在每周一到周五的6时运行，这样当你上班时就可以进行查看。

4. 你会使用什么Cmdlet来获取一个作业的结果，然后在作业队列中如何存放这些结果呢？

第16章 同时处理多个对象

PowerShell存在的主要意义在于自动化管理，这通常意味着你将会多个目标上同时执行任务。你或许希望重启多台计算机，重新配置多个服务，修改多个邮箱等。在本章，你将学到3种可以完成这些以及其他多目标任务的技术：批处理Cmdlet、WMI方法以及对象枚举。

16.1 对于大量管理的自动化

我们当然知道本书不是一本关于VBScript的书，但我们希望使用一个VBScript的例子简单阐述多目标管理的方式——Don喜欢将“批量管理”称为过去的方式（你不需要输入下面代码并运行——我们讨论的仅仅是方法，而不是结果）。

```
For Each varService in colServices
    varService.ChangeStartMode("Automatic")
Next
```

上述方法不仅仅是在VBScript中很流行，在编程的世界都很流行。下面的步骤阐述了该段代码的作用。

(1) 假设变量colServices包含多个服务。先不管colServices是如何被赋值的，因为获取服务的方式有很多。重要的是，你已经获取到服务并将其存入变量。

(2) For Each结构将会遍历或枚举所有服务，一次一个。每次都将服务存入变量varService。使用该结构，varService将会仅包含一个服务。如果colServices包含50个服务，则该循环体结构将会执行50次，每一次varService变量都会只包含这50个服务中的一个。

(3) 在循环结构中，每次都执行一个方法——在本例中是ChangeStartMode()方法——完成某些工作。

对于上述步骤，如果再思考一下，就会发现并不是一次并行执行服务的方法，而是每次只执行一个。方式和使用图形用户界面（GUI）重新配置服务并无不同。唯一的区别是代码使得计算机每次只配置一个服务，而不是人去操作。

计算机擅长执行重复操作，所以上面的过程并不是不可取的方法。但问题在于该方法需要我们给计算机提供更长、更复杂的指令。学习给予指令集所需的语言需要花费时间，这也是管理员会尝试避免VBScript和其他脚本语言的原因。

PowerShell可以使该方法重复，我们将会在本章后面展示如何去做，因为有些时候你还是需要上述方法。但利用计算机枚举对象的方式并不是使用PowerShell最高效的方式。实际上，PowerShell提供了其他两种更加易于学习和减少输入的方式，并且功能更加强大。

16.2 首选方法：“批处理”Cmdlet

正如我们在之前章节所说，很多PowerShell Cmdlet 可以接受用于操作的批，或者称之为对象集合。

比如在第6章，你已经学习过利用管道将一个Cmdlet产生的结果传输给另一个Cmdlet，比如说下面命令（请不要运行该命令——它将使你的计算机崩溃）：

```
Get-Service | Stop-Service
```

这是一个使用批处理管理的示例。在本例中，Stop-Service专门被设计用于从管道接受一个或多个服务对象，并停止服务。Set-Service、Stop-Service、Move-ADObject以及Move-Mailbox都是接受一个或多个输入对象并执行其任务或行为的Cmdlet示例。你无须像我们在之前小结VBScript中那样使用循环结构手动枚举对象。PowerShell知道如何处理批量对，并只需要更简单的语法规则。

这就是所谓的批处理Cmdlet（这是我们对它的命名，并不是官方术语），也是我们批量管理的首选方式。比如说，我们希望改变3个服务的启动模式。我们不选择VBScript方式的方法，而是采用下面这种：

```
Get-Service -name BITS,Spooler,W32Time | Set-Service -startuptype Automatic
```

从某种程度来说，Get-Service也是一种批处理Cmdlet。这是由于该命令能够从多台计算机中获取服务。假设你需要变更同样这三台计算机上的服

务:

```
Get-Service -name BITS,Spooler,W32Time -computer  
Server1,Server2,Server3 |  
Set-Service -startuptype Automatic
```

上述方法中一个潜在的问题在于，执行动作的**Cmdlet**通常不会返回表示作业状态的结果。这意味着上面两个命令都不会产生可视化结果，这非常令人不安。值得庆幸的是，这些命令通常会有一个**-passThru**参数，该参数用于打印出该命令所接受的对象。你也可以使用**Set-Service**输出其修改的服务，并使用**Get-Service**重新获取这些服务以便查看之前的命令是否生效。

下面是不同**Cmdlet**使用**-PassThru**参数的示例。

```
Get-Service -name BITS -computer Server1,Server2,Server3 |  
Start-Service -passthru |  
Out-File NewServiceStatus.txt
```

该命令将会从3台计算机列表中获取指定的服务，然后通过管道将这些服务传递给**Start-Service**。该命令不仅会启动服务，而且会将涉及的服务对象打印在屏幕上。然后这些服务对象将会通过管道传递给**Out-File**，将这些被更新对象的信息存储在文本文件中。

再重申一次：这是我们使用**PowerShell**推荐的首选方式。如果存在可以通过**Cmdlet**完成的工作，请使用**Cmdlet**。理想情况下，**Cmdlet**的作者都会选择以对象批处理的方式处理对象，但并不总是这样（**Cmdlet**作者也在学习为我们这些管理员写**Cmdlet**的最佳方式）。这是最理想的方式。

16.3 MI方式：调用WMI方法

不幸的是，总有一些任务无法通过调用**Cmdlet**完成。而且有一些我们可以通过**Windows**管理规范（**WMI**）可以操控的条目（关于**WMI**，我们将会在第14章讲解）。

注意： 我们将通过故事线的方式帮助你体验人们如何使用**PowerShell**。这会看起来有点多余，但请记住，经验本身是无价的。

比如，WMI中的Win32_NetworkAdapterConfiguration类。该类代表与网卡绑定的配置信息（网卡可以有多个配置，但目前我们假设它只有一个配置信息，这也是对于大多数计算机的常见配置）。假如说我们的目标是在计算机上所有的intel网卡上启用DHCP，但我们不希望启用RAS或其他虚拟网卡的DHCP。

我们可以以查询网卡配置开始，得到如下输出结果。

```
DHCPEnabled      : False
IPAddress        : {192.168.10.10, fe80::ec31:bd61:d42b:66f}
DefaultIPGateway :
DNSDomain       :
ServiceName      : E1G60
Description      : Intel(R) PRO/1000 MT Network Connection
Index           : 7

DHCPEnabled      : True
IPAddress        :
DefaultIPGateway :
DNSDomain       :
ServiceName      : E1G60
Description      : Intel(R) PRO/1000 MT Network Connection
Index           : 12
```

为了得到上述输出结果，我们需要查询合适的WMI类并过滤出只有描述中包含INTEL的配置。下面的代码可以完成该功能（注意在WMI过滤中以“%”作为通配符）。

```
PS C:\> gwmi win32_networkadapterconfiguration
➡-filter "description like '%intel%'"
```

动手实验：我们欢迎你跟随本章的示例执行代码。你或许需要小幅修改命令，从而获得希望的结果。比如说，你的计算机中并没有使用Intel制造的网卡，则需要将过滤条件做适当的修改。

我们在管道中包含这些配置对象信息后，我们希望启用DHCP（你可以看到其中一块网卡并没有启用DHCP）。我们或许可以找一个名称类似“Enable-DHCP”的Cmdlet。不幸的是，我们找不到该Cmdlet，因此不存在该Cmdlet。没有任何Cmdlet可以直接在批处理中和WMI对象打交道。

下一步是查看对象本身是否包含可以启用DHCP的方法为了找出结果，我们将配置对象通过管道传输给Get-Member（或者其别名Gm）：

```
PS C:\> gwmi win32_networkadapterconfiguration
➔ -filter "description like '%intel%'" | gm
```

在结果列表的开始部分，我们可以看到我们寻找的方法EnableDHCP（）：

```
TypeName: System.Management.ManagementObject#root\cimv2\Win32_NetworkAd
apterConfiguration

Name                                MemberType      Definition
----                                -
DisableIPSec                        Method
System.Management.ManagementB...
EnableDHCP                          Method
System.Management.ManagementB...
EnableIPSec                         Method
System.Management.ManagementB...
EnableStatic                        Method
System.Management.ManagementB...
```

下一步，也是很多PowerShell新手会尝试的方法，将配置对象通过管道传递给该方法：

```
PS C:\> gwmi win32_networkadapterconfiguration
➔ -filter "description like '%intel%'" | EnableDHCP()
```

不幸的是，这是无效的。你不能将对象通过管道传输给方法，你只能将其传递给Cmdlet。EnableDHCP并不是一个PowerShell的Cmdlet，而是直接附加在配置对象自身的行为。这种传统的、类似VBScript的方法和我们在本章开篇所展示给你的VBScript示例非常类似。但使用PowerShell，你能够以更简单的方式改成该任务。

虽然没有名为Enable-DHCP的“批处理”Cmdlet，但可以使用名为Invoke-WmiMethod这个通用的Cmdlet。该Cmdlet特别设计用于接受一批WMI对象，比如说我们的Win32_NetworkAdapter Configuration对象，并调用附加在这些对象上的某个方法。下面是我们运行的命令。

```
PS C:\> gwmi win32_networkadapterconfiguration
➔ -filter "description like '%intel%'" |
➔ Invoke -WmiMethod -name EnableDHCP
```

你需要记住如下几条:

- 方法名称后无须加括号。
- 方法名称不区分大小写。
- **Invoke-WmiMethod**一次只能接收一种类型的WMI对象。在本例中，我们只发送给**Win32_NetworkAdapterConfiguration**一种对象，这意味着命令可以如预期产生效果。当然也可以一次发送多个对象（实际上，这是重点），但所有的对象都必须是同一类型。
- 你可以使用针对**Invoke-WmiMethod**方法加上**-WhatIf**和**-Conifrm**参数。但直接由对象调用方法时，无法使用这些参数。

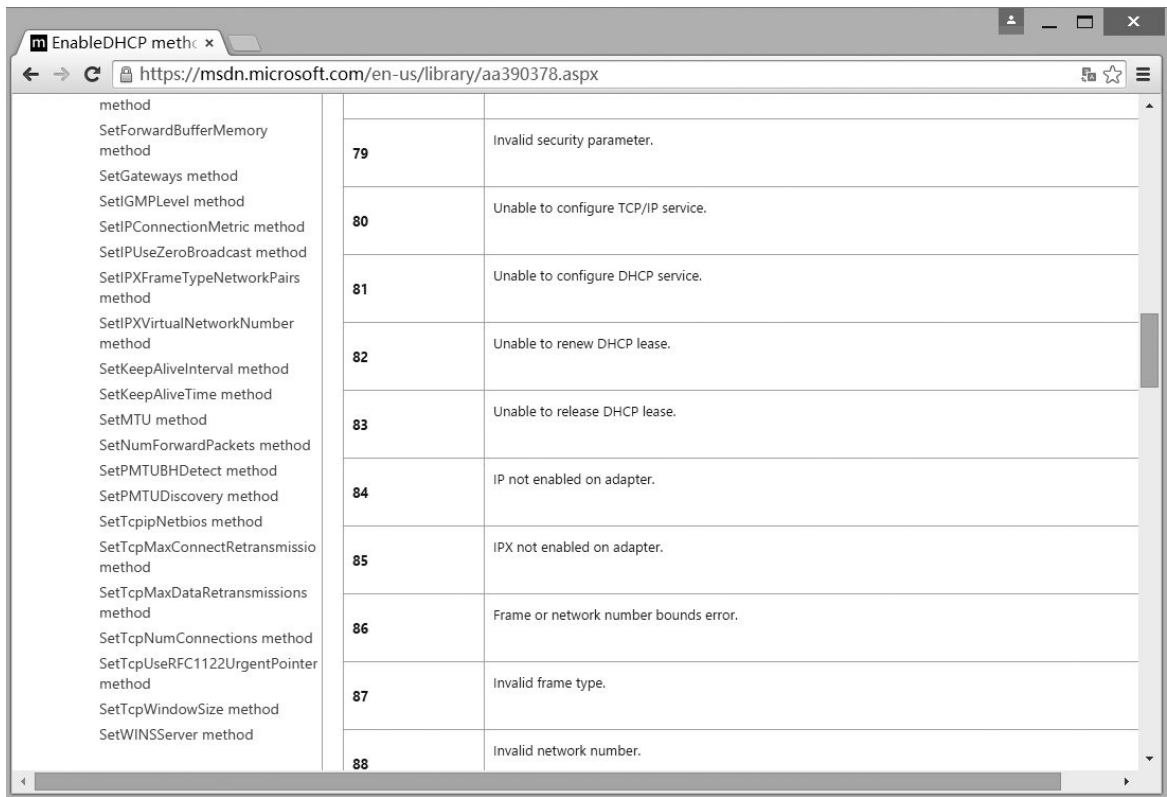
Invoke-WmiMethod的输出结果有点让人困惑。**WMI**总是产生结果对象，并包含大量系统对象（名称以两个下划线开始）。在本例中，命令产生如下输出结果。

```
__GENUS           : 2
__CLASS           : __PARAMETERS
__SUPERCLASS      :
__DYNASTY         : __PARAMETERS
__RELPATH         :
__PROPERTY_COUNT  : 1
__DERIVATION      : {}
__SERVER          :
__NAMESPACE       :
__PATH           :
ReturnValue       : 0

__GENUS           : 2
__CLASS           : __PARAMETERS
__SUPERCLASS      :
__DYNASTY         : __PARAMETERS
__RELPATH         :
__PROPERTY_COUNT  : 1
__DERIVATION      : {}
__SERVER          :
__NAMESPACE       :
__PATH           :
ReturnValue       : 84
```

上述结果唯一有用的信息是一个没有以双下划线开头的属性：**ReturnValue**。该数字告诉我们操作的结果。通过Google搜索“Win32_NetworkAdapterConfiguration”出现文档页，我们通过单击**EnableDHCP**方法找到可能返回的值以及其代表的意义。图16.1展示了我们发现的结果。

0表示成功，而84表示该网卡配置中未启用IP，因此DHCP无法启用。但该值对应哪一个网卡配置呢？这很难说。这是由于输出结果并没有告诉你是由哪一个配置对象产生。虽然令人遗憾，但这就是WMI的工作机制。



The screenshot shows a web browser window with the address bar displaying <https://msdn.microsoft.com/en-us/library/aa390378.aspx>. The page content is a table listing return values for the `EnableDHCP` method. The table has three columns: 'method', 'return value', and 'description'. The 'method' column lists various network configuration methods, and the 'return value' column lists error codes from 79 to 88. The 'description' column provides the meaning of each error code.

method	return value	description
SetForwardBufferMemory method	79	Invalid security parameter.
SetGateways method	80	Unable to configure TCP/IP service.
SetGMPLevel method	81	Unable to configure DHCP service.
SetIPConnectionMetric method	82	Unable to renew DHCP lease.
SetIPUseZeroBroadcast method	83	Unable to release DHCP lease.
SetIPXFrameTypeNetworkPairs method	84	IP not enabled on adapter.
SetIPXVirtualNetworkNumber method	85	IPX not enabled on adapter.
SetKeepAliveInterval method	86	Frame or network number bounds error.
SetKeepAliveTime method	87	Invalid frame type.
SetMTU method	88	Invalid network number.
SetNumForwardPackets method		
SetPMTUBHDetect method		
SetPMTUDiscovery method		
SetTcpipNetbios method		
SetTcpMaxConnectRetransmissions method		
SetTcpMaxDataRetransmissions method		
SetTcpNumConnections method		
SetTcpUseRFC1122UrgentPointer method		
SetTcpWindowSize method		
SetWINSServer method		

图16.1 找WMI方法返回值的结果

当你有一个WMI对象包含可执行的方法时，大多可以使用**Invoke-WmiMethod**。该命令对于远程计算机同样有效。我们的基本原则是“如果你可以使用**Get-WmiObject**获取对象，则也能够使用**Invoke-WmiObject**执行它的方法”。

当你回忆第14章所学内容时，你会发现**Get-WmiObject**与**Invoke-WmiMethod**都是“遗留”用于操作WMI的Cmdlet；这两个命令的接替者为**Get-CimInstance**和**Invoke-CimMethod**。它们的工作方式或多或少有些相同：

```
PS C:\> Get-CimInstance -classname win32_networkadapterconfiguration  
➔ -filter "description like '%intel%'" |  
➔ Invoke-CimMethod -methodname EnableDHCP
```

在第14章中，我们提供了何时使用WMI或CIM的建议，该建议在此同样适用：虽然WMI需要难以穿透防火墙的RPC网络通信，但WMI能够适用的计算机数量最多（当前来说）；CIM只需要更新更简单的WS-MAN通信，但在老版本的Windows默认情况下，WS-MAN并没有安装。

但请等一下，还有一件事，我们在本小节讨论了WMI，并在第14章中提到微软做了很多工作，也就是将WMI功能封装进了Cmdlet，以至于无意中对你隐藏了WMI的存在。请尝试在PowerShell中运行Help Set-NetIPAddress。在较新版本的Windows中，你将会发现这个强大的Cmdlet掩盖了大量底层WMI的复杂性。我们可以使用该Cmdlet变更IP地址，而无需一大堆WMI。这是一个真实的教训：即使你在网上阅读了关于该主题的一些资料，也并不意味着新版本的PowerShell没有提供更好的方式。大多数发布在网上的资料都是基于PowerShell v1和v2，但v3和更新的版本中提供的Cmdlet至少比之前的好4~5倍。

16.4 后备计划：枚举对象

不幸的是，我们遇到的一些情况是Invoke-WmiObject无法执行某个方法——执行时不断返回奇怪的错误信息。我们还遇到的一些情况是虽然某个Cmdlet可以产生对象，但我们知道并没有可以通过管道接收这些对象并进行操作的批处理Cmdlet。无论是上述哪种情况，你依然可以完成任务，但你必须回到传统的VBScript风格的方法来指挥计算机枚举对象并一次执行一个对象。PowerShell提供了两种方法：第一种是使用Cmdlet，另一种是使用脚本结构。我们在本章主要关注第一种技术，并在第21章阐述第二种。在第21章中，我们将会深入PowerShell内置的脚本语言。

我们使用Win32_Service这个WMI类作为示例。更详细地说，我们将使用Change()方法。这是一个可以一次性变更某个服务中多个元素的复杂方法。图16.2展示了其在线文档（我们通过搜索“Win32_Service”找到该列面并导航的Change方法页）。

通过阅读该页，你会发现无须为该方法的每一个参数赋值。你可以将你希望忽略的参数指定为Null（PowerShell中有一个特殊的内置\$null变量）。

对于本例来说，我们希望变更服务的启动密码，也就是第8个参数。为了完成该工作，我们需要将前7个参数指定为\$null。这意味着我们的方法执行代码会类似如下。

```
Change($null, $null, $null, $null, $null, $null, $null, "P@ssw0rd")
```

顺便提一下，无论是Get-Service还是Set-Service，都无法显示或设置某个服务的登录密码。但WMI可以完成该工作，所以我们使用WMI。

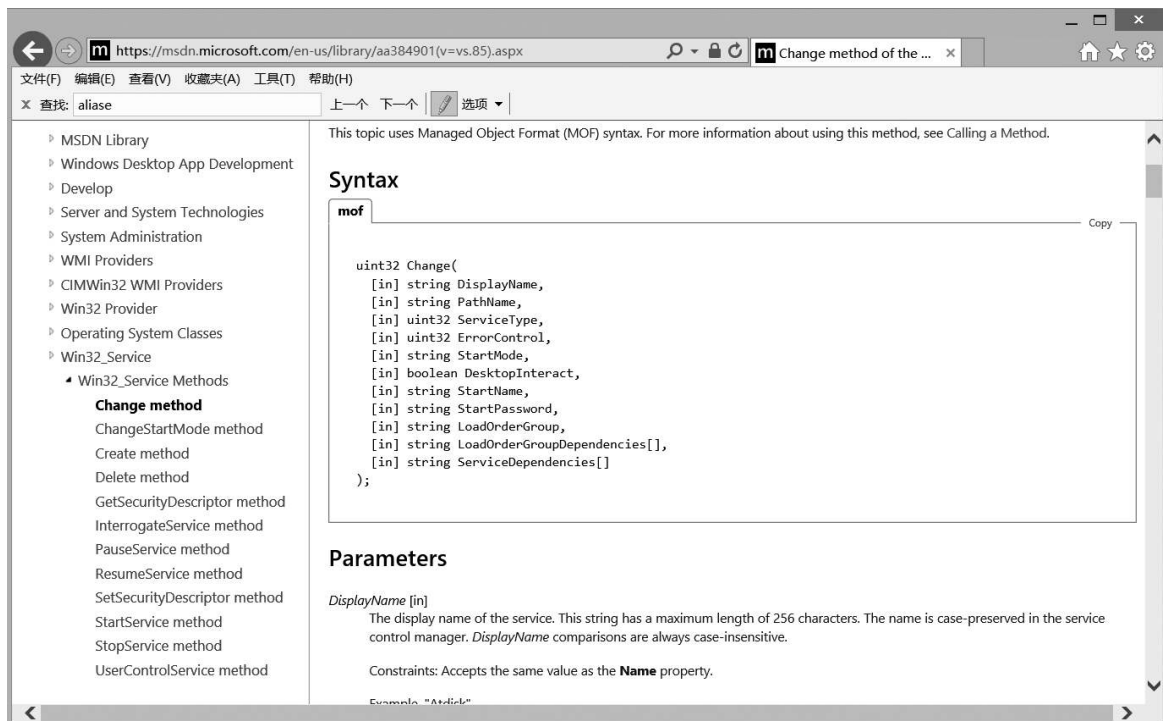


图16.2 in32_Service的Change()方法的文档页

由于我们无法使用首选的Set-Service这个批处理Cmdlet，让我们尝试第二种方式，也就是使用Invoke-WmiMethod。该Cmdlet包含一个参数：-ArgumentList，可以利用该参数为方法指定参数。下面的示例是我们进行的尝试以及接收的结果。

```
PS C:\> gwmi win32_service -filter "name = 'BITS'" | invoke-wmimethod -
name
change -arg $null,$null,$null,$null,$null,$null,$null,"P@ssw0rd"
Invoke-WmiMethod : Input string was not in a correct format.
At line:1 char:62
+ gwmi win32_service -filter "name = 'BITS'" | invoke-wmimethod <<<< -
```

```
name change -arg $null,$null,$null,$null,$null,$null,$null,"P@ssw0rd"  
+ CategoryInfo          : NotSpecified: (:) [Invoke-WmiMethod],  
FormatException  
+ FullyQualifiedErrorId :  
System.FormatException,Microsoft.PowerShell  
Commands.InvokeWmiMethod
```

注意： 我们这里使用的是Get-WmiObject，但Get-CimInstance的语法与其几乎相同。

此时，我们必须做出决定。有可能我们没有用正确的方式运行命令，所以我们必须决定是否花一些时间找出原因。还有一种可能是Invoke-WmiMethod与Change()方法的兼容性存在问题。如果是这个问题的话，就需要我们花费大量时间在我们无法控制的事情上。

对于这种情况，我们通常会尝试其他方式：我们将会要求计算机（好吧，是Shell）枚举所有服务对象，每次一个，并对每个对象执行Change()方法。我们将使用ForEach-Object这个Cmdlet完成这项工作。

```
PS C:\> gwmi win32_service -filter "name = 'BITS'" | foreach-object  
{$_ .change($null,$null,$null,$null,$null,$null,$null,"P@ssw0rd") }  
  
__GENUS           : 2  
__CLASS           : __PARAMETERS  
__SUPERCLASS      :  
__DYNASTY         : __PARAMETERS  
__RELPATH         :  
__PROPERTY_COUNT  : 1  
__DERIVATION      : {}  
__SERVER          :  
__NAMESPACE       :  
__PATH            :  
ReturnValue       : 0
```

在文档中，我们发现ReturnValue为0表示成功。这意味着我们已经实现了目标。但让我们可以将命令格式化得更美观，更仔细地看这个命令：

```
Get-WmiObject Win32_Service -filter "name = 'BITS'" |  
➔ ForEach-Object -process {  
➔   $_.change($null,$null,$null,$null,$null,$null,$null,"P@ssw0rd")  
➔ }
```

该命令中包含很多内容。第一行看起来很合理：我们使用Get-WmiObject获取所有满足过滤条件的Win32_Service实例，也就是名称包含“BITS”的服务（照例，我们选择BITS服务是由于相比其他服务来说，该服务并没有那么重要，该服务停止运行不会导致计算机崩溃）。然后将Win32_Service对象传递给ForEach-Object这个Cmdlet。

让我们把之前示例中的代码分解为模块：

- 首先，你将看到Cmdlet名称：ForEach-Object。
- 接下来，使用-Process参数指定脚本段。我们原先并没有输入-Process的参数名称，这是由于该参数为位置参数。但脚本段中，所有在花括号中的代码都是-Process参数的值。所以我们接下来将参数名称包含在内，并更好地格式化，以方便阅读。
- ForEach-Object将会对于每一个通过管道传输给ForEach-Object的对象执行脚本段。每次脚本段执行后，下一个通过管道传输进来的对象都会被置于特殊的\$_容器。
- 通过在\$_后输入一个“.”，告诉Shell我们需要访问当前对象的属性或方法。
- 在示例中，我们访问Change()方法。注意，方法的参数以逗号分隔列表方式存在，并被包在括号内。我们使用\$null作为我们不希望变更的参数传入，并将新密码作为第8个参数。该方法可以接受更多参数，但由于我们不希望修改第9个、第10个或第11个参数，我们可以完全忽视它。（我们也可以将最后三个参数指定为\$null。）

我们当然传达了一个复杂的语法。图16.3将帮助你分解它。

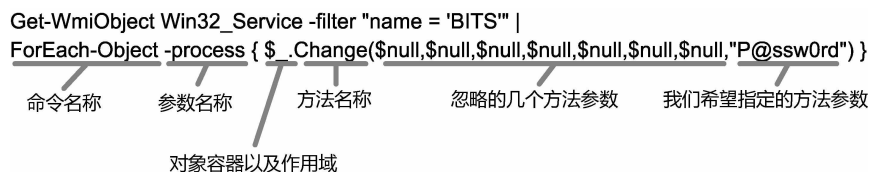


图16.3 分解ForEach-Object Cmdlet

你可以对WMI方法使用完全同样的模式。为什么你从不使用Invoke-WmiMethod来替代上面的方法呢？好吧，该命令通常会起作用，并更容易输入和阅读。但如果你更倾向于只记住一种方式，那就是ForEach-Object方式。

我们不得不警告你，在网上看到的示例可能或更难以阅读。PowerShell专家更倾向于使用别名、位置参数以及最短的参数名称，这会降低可读性（但节省输入）。下面是同样的命令，但以最短的形式。

```
PS C:\> gwmi win32_service -fi "name = 'BITS'" |  
➔ % {$_change($null,$null,$null,$null,$null,$null,$null,"P@ssw0rd") }
```

让我们查看一下我们所做的变更：

- 我们使用Gwmi而不是Get-WmiObject。
- 我们将-filter简写为-fi。
- 我们使用%这个别名代替ForEach-Object。是的，百分号符号是该Cmdlet的别名。我们发现该别名难以阅读，但很多人这么用。
- 我们再次删除了-Process的参数名称，这是由于该参数是位置参数。

在博客或其他地方分享脚本时，我们并不喜欢使用别名和简写的参数名称。这是由于该方法使得其他人难以阅读。如果你将一些代码存入脚本，花费一些时间将代码输入完整是值得的（或者使用Tab自动补全功能让Shell帮你输入）。

如果你希望使用本例，下面是一些你希望改变的地方（见图16.4）。

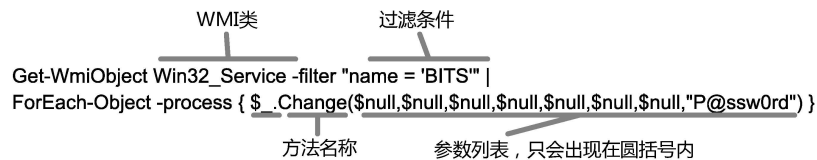


图16.4 可以对之前示例所做的变更，以便执行不同的WMI方法

- 你或许希望改变WMI名称或者过滤条件，以取得你希望取得的WMI对象。
- 你可以将方法名称从Change修改为你希望执行的方法名称。
- 你可以修改方法的参数（也被称为“argument”）列表为任何你的方法期望的参数列表。参数列表总是一个逗号分隔的列表，并包裹在圆括号内。对于没有任何参数的方法，圆括号内可以为空，比如我们在本章开篇介绍的EnableDHCP()方法。

这是否是实现我们目标的最佳方式？通过查看Set-Service的帮助文档，我们发现该命令并没有提供修改密码的方式，而Get-Wmi-Object和Get-CimInstance这两个命令都可以完成该功能。这使得我们可以做出总结：即使是PowerShell v3，对于这个任务，WMI依然是一种值得使用的方式。

16.5 常见误区

我们本章中所涵盖的技术是PowerShell中最难的技术，这些技术是在我们班级中导致最多困惑的技术。让我们来看一些学生们经常遇到的问题，并提供一些替代的阐述方式。我们希望能够帮助你避免同样的问题。

16.5.1 哪一种正确的方式

我们使用术语“批处理Cmdlet”或“行为Cmdlet”指代那些针对一组对象或对象集合操作的Cmdlet。你可以将一组对象发送给Cmdlet并由Cmdlet对循环进行处理，而不是指示计算机“遍历列表中的东西，并对列表中的每一个东西执行某些行为”。

微软在其产品中提供这类Cmdlet方面做得越来越好，但并没有100%覆盖所有功能（很可能以后很多年也覆盖不了，这是由于存在大量复杂的微软产品）。但当存在一个我们所需的Cmdlet时，我们更倾向使用Cmdlet。即便如此，其他PowerShell的开发人员根据他们先学到的和他们更容易记住的倾向于选择其他替代办法。下面所有的命令实现的功能完全相同。

<code>Get-Service -name *B* Stop-Service</code>	← ❶ 批处理 cmdlet
<code>Get-Service -name *B* ForEach-Object { \$_.Stop() }</code>	← ❷ ForEach-Object
<code>Get-WmiObject Win32_Service -filter "name LIKE '%B%'" Invoke-WmiMethod -name StopService</code>	← ❸ WMI
<code>Get-WmiObject Win32_Service -filter "name LIKE '%B%'" ForEach-Object { \$_.StopService() }</code>	WMI和 ForEach-Object
<code>Stop-Service -name *B*</code>	← ❹ Stop-Service

让我们来看一下每种方式的工作机制：

- 第一种方式是使用批处理Cmdlet❶。这里，我们使用Get-Service获取所有名称包含“B”的服务，并停止这些服务。
- 第二种方式类似。但使用ForEach-Object来替代批处理Cmdlet，并要求每个服务执行Stop()方法❷。
- 第三种技术是使用WMI，而不是Shell的原生管理Cmdlet❸。我们接收到需要的服务（也就是名称包含字母“B”的服务），并通过管道传递给Invoke-WmiMethod。我们告诉该命令调用StopService方法，这是WMI服务对象使用的方法名称。
- 第四种方式是使用ForEach-Object而不是Invoke-WmiMethod实现完全相同的工作❹。这种方式结合了方式2和方式3，并不是一种全新的方式。
- 第五种方式是直接使用Stop-Service❹，但其-Name参数（在PowerShell v3）接受通配符。

见鬼！其实还有第六种方式——使用PowerShell的脚本语言完成工作。你将会发现PowerShell中每一项工作都可以使用多种方式完成，且没有哪一种是错误的。某些方式比其他方式更易于学习、记忆以及重复，这也是为什么我们按照所做的顺序关注我们可以使用的技术。

我们的例子还阐述了使用原生Cmdlet和WMI的重要区别。

- 原生Cmdlet过滤条件通常使用“*”作为通配符，而WMI过滤使用百分比符号（%）——请不要将百分比符号和ForEach-Object别名搞混。这个百分比符号是封装在Get-WmiObject的-filter参数内，它并不是一个别名。
- 原生对象通常和WMI有同样的功能，但语法或许会有不同。在本例中，由Get-Service产生的ServiceController对象有Stop()方法；而我们通过WMI的Win32_Service类访问同样的对象时，方法名称变为StopService()。
- 原生过滤通常使用原生的比较操作符，比如说-eq；WMI使用类似编程语言风格的操作符，比如说=或者Like。

我们该使用哪一种方式？这无所谓，因为并没有一种所谓“对”的方式。你甚至会根据环境以及Shell能够提供给你的功能混合使用这两种方式。

16.5.2 WMI方法与Cmdlet对比

你何时该使用WMI方法或Cmdlet来完成一个任务呢？这个选择十分简单。

- 如果你通过Get-WmiObject获取对象，你将需要通过使用WMI方法来执行行为。你可以使用Invoke-WmiMethod或ForEach-Object方式执行方法。
- 如果你通过非Get-WmiObject的方式获取对象，你将需要对获取到的对象使用原生Cmdlet。除非你获取到的对象只有方法而没有能够完成任务所需的Cmdlet，你可能会使用ForEach-Object方式执行方法。

注意，到这里的最低标准是ForEach-Object：它的语法或许是最难的，但你可以使用它完成几乎所有你需要完成的工作。

永远无法将任何对象通过管道传递给一个方法。你只能利用管道将一个Cmdlet产生的对象传递给另一个Cmdlet。如果不存在完成任务所需的Cmdlet，但存在这样的方法，那么你就可以将其通过管道传递给ForEach-Object并执行对象的方法。

例如，假设你通过Get-Something这个Cmdlet获取到对象，你希望删除这些对象，但不存在Delete-Something或Remove-Something这样的Cmdlet。但该对象包含Delete方法，那么你就可以这么做：

```
Get-Something | ForEach-Object { $_.Delete() }
```

16.5.3 方法文档

请总是记住，通过管道将对象传递给Get-Member，可以查看该对象包含的方法。我们在此使用Get-Something这个Cmdlet作为示例。

```
Get-Something | Get-Member
```

PowerShell的内置帮助系统并未记录WMI方法的文档。你需要使用搜索引擎（通常搜索WMI类的名称）来找到WMI方法的指南和示例。你也无法在PowerShell内置的帮助系统中找到非WMI对象放的文档。比如说，如果你获取一个服务对象的成员列表，你将会发现存在名称为Stop和Start的方法。

TypeName: System.ServiceProcess.ServiceController		
Name	MemberType	Definition
----	-----	-----
Name	AliasProperty	Name = ServiceName
RequiredServices	AliasProperty	RequiredServices =
ServicesDepe...		
Disposed	Event	System.EventHandler Disposed(Sy...
Close	Method	System.Void Close()
Continue	Method	System.Void Continue()
CreateObjRef	Method	System.Runtime.Remoting.ObjRef ...
Dispose	Method	System.Void Dispose()
Equals	Method	bool Equals(System.Object obj)
ExecuteCommand	Method	System.Void ExecuteCommand(int ...
GetHashCode	Method	int GetHashCode()
GetLifetimeService	Method	System.Object GetLifetimeService()
GetType	Method	type GetType()
InitializeLifetimeService	Method	System.Object
Initializelifetim...		
Pause	Method	System.Void Pause()
Refresh	Method	System.Void Refresh()
Start	Method	System.Void Start(), System.Voi...
Stop	Method	System.Void Stop()
ToString	Method	string ToString()

WaitForStatus	Method	System.Void WaitForStatus(Syste...
---------------	--------	------------------------------------

如果希望找到该对象的文档，请重点关注Type Name，在本例中也就是System.Service Process.ServiceController。在搜索引擎中搜索完整的类型名称，你通常可以找到完整的官方开发文档，并可以根据文档找出你所需的特定方法的文档。

16.5.4 ForEach-Object相关误区

ForEach-Object这个Cmdlet的语法中包含大量标点符号，再加上方法自带的语法，会导致出现难以阅读的命令行。我们准备了一些小技巧帮你打破僵局。

- 多使用ForEach-Object的完整名称，而不是使用%或ForEach这样的别名。完整名称更易于阅读。如果你使用别人写的示例，请将别名替换为完整名称。
- 花括号内的代码段对于每一个通过管道传入的对象执行一次。
- 在代码段内，\$_代表通过管道传入的对象之一。
- 使用\$_本身控制所有通过管道传入的对象；使用\$_后的加“.”控制单独的方法或属性。
- 即使方法不需要任何参数，方法名称之后也总是跟随圆括号。当需要参数时，通过逗号将参数分隔放在括号内。

16.6 动手实验

注意： 对于本次动手实验来说，你需要运行PowerShell v3或更新版本PowerShell的计算机。

尝试回答接下来的问题并完成指定任务。这是一个重要的实验，因为该实验需要利用你在之前章节所学的技巧。随着你读完本书剩下的内容，你还需要不断巩固这些技巧。

1. 哪一个ServiceController对象（由Get-Service产生）的方法将会暂停服务，而不是完全停止服务？
2. 哪一个Process对象的方法（由Get-Process产生）可以终止指定的进程？

3. 哪一个WMI对象Win32_Process的方法将会终结一个给定进程？

4. 写4个不同命令，利用该命令可以终结所有名称为“Notepad”的进程。在此假设多个进程以同样的进程名称运行。

第17章 安全警报

现在，你已经知道PowerShell是多么强大，但是你会突然意识到一个问题：存在这些强大的功能会不会造成一些安全隐患？答案是“可能会”。在本章中，我们会帮助你了解PowerShell将如何影响你环境的安全，同时会讲解如何配置PowerShell才能取得安全和强大功能上的平衡。

17.1 保证Shell安全

自从2006年年底PowerShell发布以来，微软在安全和脚本方面并没有取得很好的名声。毕竟那个时候，VBScript和Windows Script Host(WSH)是两个最流行的病毒和恶意软件的载体，它们经常成为臭名昭著的“I Love You”“Melissa”等其他病毒的攻击点。当PowerShell团队宣布他们创造了一种新的、能提供前所未有强大的功能与可编程能力的命令行Shell语言时，我们认为，警报来临，人们将对这种新的命令行Shell避之不及。

但是，没关系。PowerShell是在比尔·盖茨先生在微软发起的一个“可信计算计划”之后才进行开发的。在微软公司内部，该计划产生了很积极的效果：每个产品部门都要求配备一名资深软件安全专家，该专家会参与到设计会议、代码复审等工作中。该专家被称为产品的“安全伙伴”（并不是我们编造的）。PowerShell产品的“安全伙伴”是经由微软出版的圣经《编写安全代码》(Writing secure code)的其中一位作者，该书描写了如何编写不易受攻击者利用的软件。我们可以保证PowerShell与其他产品一样都是安全的——至少默认情况下，都是安全的。当然，你也可以修改这些默认值，但是当你进行操作时，请在考虑功能之外，也要注意安全问题。这也就是本章要帮你完成的事情。

17.2 Windows PowerShell的安全目标

我们需要明确，当谈及安全时，PowerShell会做什么，又不会做什么；最好的办法是列出一些PowerShell的安全目标。

首先，PowerShell不会给处理的对象应用任何额外的权限。也就是说，PowerShell仅会在你已拥有的权限主体下处理对象。比如，如果通过图形用户界面操作，你没有在活动目录中创建新用户的权限，那么在PowerShell中你也无法创建该用户。总体来说，PowerShell仅仅是你使用当前权限来完成某些操作的一种实现方式而已。

其次，PowerShell无法绕过既有的权限。比如，想给你的用户部署某个脚本，并希望该脚本能完成某些操作——正常情况下这些用户会由于权限不足无法完成的某些操作，那么最后，该脚本也不能运行。如果希望用户可以完成某些操作，那么必须给他们赋予对应的权限；PowerShell仅能完成这些用户凭借现有权限执行命令或者脚本可以完成的工作。

设计PowerShell安全系统的目的并不是为了阻止用户在正常的权限下输入并运行某些命令。该思想是使得欺骗用户输入很长的、较为复杂的命令变得更加困难，因此PowerShell不会应用超过该用户当前拥有的权限之外的安全设置。从过去的经验我们知道，欺骗用户运行一段可能包含恶意代码的命令是非常简单的事。这也就是为什么PowerShell的大部分安全设置都被设计为阻止用户运行一些未知的脚本。“意外”这个部分是非常重要的：PowerShell的安全并不旨在阻止一个已确定用户运行脚本，只是为了阻止用户被欺骗运行来自不受信任来源的脚本。

补充说明：

下面讲到的内容超出本书范围，但是还是希望你能知道存在其他一些方法可以使得用户在其他凭据（而非自有凭据）下运行某些命令。通常称这种技术为脚本封装。它是一些商业脚本开发环境的一个特性，比如SAPIEM PrimalScript(www.PrimalTools.com)。

创建一个脚本之后，你可以使用打包程序将这个脚本放入到一个可执行文件(.EXE)中。这并不是编码学中的编译过程：这个可执行文件并不是独立的，它需要在PowerShell安装之后才能执行。你也可以通过配置打包程序，将可用的凭据加密到可执行文件中。这样，如果有人运行该可执行文件，其中的脚本会在指定的凭据下被执行，而不依赖当前用户的凭据。

当然，被封装的凭据也不是百分之百安全。被封装的文件中都会包含用户名以及对应密码，尽管大部分打包程序都会进行用户及密码的加密。准确地说，针对大部分用户而言，他们都无法发现用户名以及对应的密码；但是针对一个熟练的加密专家来说，破解出用户名以及密码是很简单的一件事。

PowerShell的安全并不是针对恶意软件的防护。一旦在你的系统上存在恶意软件，那么恶意软件可以做你权限范围内的任何事情。它可能使用PowerShell去执行一些恶意命令，也有可能非常轻易地使用几十种技术来损坏你的电脑。一旦在你的系统中存在恶意软件，那么你就被“挟持”了。当然，PowerShell也并不是第二道防御系统。此时，首先你需要杀毒软件来阻止恶意软件进入你的系统。对大部分人而言，可能忽略这样一个重要的概

念：即使恶意软件可能借助PowerShell去完成一些危害行为，也不应该将恶意软件问题归咎于PowerShell。杀毒软件必须阻止恶意软件运行。再次申明，PowerShell设计出来并不是为了保护一个已经受损的系统。

17.3 执行策略和代码签名

PowerShell中第一个安全措施是执行策略。执行策略是用来管理PowerShell执行脚本的一种计算机范围的设置选项。正如本章前面所讲，该策略主要用作防止用户被注入，从而执行一些非法脚本。

17.3.1 执行策略设置

默认设置是Restricted，该策略会阻止正常脚本的运行。也就是说，默认情况下，你可以使用PowerShell进行交互式执行命令，但是你不能使用PowerShell执行脚本。如果你尝试执行脚本，你会得到下面的错误：

```
无法加载文件 C:\test.ps1，因为在此系统中禁止执行脚本。有关详细信息，请参阅 "get
-help about_signing"。
所在位置行:1 字符: 11
+ .\test.ps1 <<<<
+ CategoryInfo          : NotSpecified: (:) [], PSSecurityException
+ FullyQualifiedErrorId :RuntimeException
```

你可以通过运行Get-ExecutionPolicy命令来查看当前的执行策略。另外，如果你想修改当前的执行策略，可以采用下面三种方式之一：

- 运行Set-ExecutionPolicy命令。该命令会修改Windows注册表中的HKEY_LOCAL_MACHINE部分，但是需要在管理员权限下才能执行该命令，因为一般用户没有修改注册表的权限。
- 使用组策略对象（GPO）。从Windows Server 2008 R2开始，Windows PowerShell相关的设置已经内置在系统中。对于老版本的域控制器，你可以通过下载一个ADM模板来扩展当前组策略功能。你可以在网站<http://mng.bz/U6tJ>上找到该模板。当然也可以通过访问网站<http://download.microsoft.com>，之后在搜索框中输入PowerShell ADM进行查找。

如图17.1所示，我们可以在“本地计算机策略”>用户配置>管理模板>Windows组件>Windows PowerShell中找到PowerShell的设置选项。图17.2展示了我们将该策略设置为“启用”的状态。当通过组策略对象来配

置时，组策略中的设定会覆盖本地的任何设置值。实际上，如果你试图运行Set-ExecutionPolicy，命令可以正常执行，但是会返回一个警告。该警告会告知由于组策略覆盖的原因，你新修改的设定值不会起作用。



图17.1 Windows PowerShell设置在组策略中的位置



图17.2 在组策略对象中修改Windows PowerShell的执行策略

- 通过手动运行PowerShell.exe，并且给出-ExecutionPolicy的命令行开关参数。如果采用这种方式，那么命令中指定的执行策略会覆盖本地任何设置和组策略中的设置值。

你可以将执行策略设置为5种值（请注意：组策略对象中包含下面列表中的3个选项）。

- **Restricted**——这是默认选项，除微软提供的一部分配置PowerShell的默认选项的脚本外，不允许执行其他任何脚本。这些脚本中附带微软的数字签名。如果对数字签名进行修改，那么这些脚本就再也无法运行了。
- **AllSigned**——经过受信任的证书颁发机构（CA）设计的数字证书签名之后的任意脚本，PowerShell均可执行。
- **RemoteSigned**——PowerShell可以运行本地任何脚本，同时经由受信任的CA签发的数字证书签名之后的远程脚本也可以被执行。“远程脚

本”是指存在于远端计算机上的脚本，经常通过通用命名规则（UNC）方式来访问这些脚本。我们也会将那些来自于网络上的脚本称为“远程脚本”。Internet Explorer、Firefox和Outlook中提供的可下载的脚本，我们均可视为来自网络的脚本。在某些版本的Windows中，会区分网络路径以及UNC路径。在这些场景中，本地网络中的UNC都不会认为是“远程”。

- **Unrestricted**——所有的脚本都可以运行。我们不是很喜欢或者不建议这个设置选项，因为该设置选项无法提供足够的保护功能。
- **Bypass**——这个特殊的设定主要是针对应用程序开发人员，他们会将PowerShell嵌入到他们的应用程序中。这个设定值会忽略已经配置好的执行策略，应当仅在主机应用程序提供了自身的脚本安全层时才使用该选项。

等等，什么？

你是否注意到，我们可以在组策略对象中设置一种执行策略，但是也可以使用PowerShell.exe的一个参数来覆盖该设定？通过GPO控制的设定能轻易被覆盖，这样有什么好处呢？这里主要是体现了执行策略被设计出来的一个目的：防止不知情的用户无意中运行一些匿名脚本。

执行策略并不是为了阻止用户去运行某个已知的脚本。如果真是这样，那么执行策略就不算是一种安全设置。

事实上，使用PowerShell脚本来传播恶意软件的人并不多。一个聪明的恶意软件开发者不会使用PowerShell作为介质，而是会直接去访问.Net框架的功能。

微软强烈建议在执行脚本时使用RemoteSigned执行策略，并且仅在需要执行脚本的机器上采用该策略。根据微软的建议，其他计算机应当继续保持Restricted的执行策略。微软解释道：RemoteSigned策略在安全性和功能之间取得了较好的平衡；AllSigned相对更严格，但是它要求所有脚本都需要被数字签名。PowerShell社区作为一个整体是更开放的，在到底哪种执行策略较优的问题上，存在大量的意见。就当前而言，我们会采纳微软的建议。当然，如果你有兴趣，你可以自己研究该主题。

现在，我们可以深入讨论数字签名的话题了。

注意：多个专家，包括微软的一些开发人员，都建议使用Unrestricted作为执行策略。他们觉得该功能并没有提供一个安全层，并且你也不应该相信该设置可以将任何危险的行为隔离开。

17.3.2 数字代码签名

数字代码签名，简称为代码签名，是指将一个密码签名应用到一个文本文件的过程。签名会显示在文件末端，并且类似下面的形式。

```
<!-- SIG # Begin signature block -->
<!-- MIIXXAYJKoZIhvcNAQcCoIIXTTCCF0kCAQExCzAJBgUrDgMCGGUAMGkGCisGAQQB -
->
<!-- gjcCAQSGwzBZMDQGCisGAQQBgjcCAR4wJgIDAQAABBAfzDtgWUsITrck0sYpfvNR -
->
<!-- AgEAAgEAAgEAAgEAAgEAMCEwCQYFKw4DAhoFAAAQUJ7qroHx47PI1dIt4lBg6Y5Jo -
->
<!-- UVigghIxMIIIEYDCCA0ygAwIBAgIKLqsR3FD/XJ3LwDAJBgUrDgMCHQUAMHAXKzAp -
->
<!-- YjcCn4FqI4n2XG0PsFq70ddgjFWEGjP105iggggiX4uzLLehpcur2iC2vzAZhSAU -
->
<!-- DSq8UVRB4F4w45IoaYfBc0LzP6v0gEJydg4wggR6MIIDYqADAgECAgphBieBAAAA -
->
<!-- ZngnZui2t++Fuc3uqv0SpAtZIikvz0DZVgQbdrVtZG1KVNvd8d6/n4PHgN9/TAI3 -
->
<!-- an/xvmG4PNGSdjy8Dcbb5otiSjgByprAttPPf2EKUQrFPzREgZabAatwMKJbeRS4 -
->
<!-- kd6Qy+RwkCn1UWIeaChbs0LJhix0jm38/pLCC0o1nL79E1sxJumCe6GtqjdW0IBn -
->
<!-- KKe66D/GX7eGrfCVg2Vzgp4gG7fHADFEh30cIvoILWc= -->
<!-- SIG # End signature block -->
```

签名中包含了两部分重要信息：一是列出了对脚本签名的公司或者组织；二是包含了对脚本的加密副本，并且PowerShell可以解密该副本。要理解这部分信息的工作原理，你需要了解一些背景知识。当然，这部分背景知识也会帮助在你的环境中决定该采用何种安全策略。

在创建一个数字签名之前，你需要拥有一个代码签名的证书。这些证书也被称为第三类证书。这些证书均由商业CA签发，比如Cybertrust、GoDaddy、Thawte、VeriSign等公司。当然，如果可能的话，你也可以从公司内部的公钥基础设施（PKI）中获取到该证书。正常情况下，第三类证书仅会签发给公司或者组织，而不会发给个人。当然，在公司内部可以签发给个人。在签发证书之前，CA需要验证接收方的身份——证书是类似一种数字识别卡，该卡上列出了持有者的姓名以及其他详细信息。比如，在签发证书给XYZ公司之前，CA需要验证XYZ公司的授权代表人提交了该请求。在整个安全体系中，验证过程是最重要的环节，你应当仅信任能出色完成验证申请证书的公司身份工作的CA。如果你对一个CA的验证流程不熟悉，那么你不应该信任该CA。

应当在Windows的IE浏览器属性控制面板（也可以在组策略中配置）中配置信任关系。在该控制面板中，选择到Content标签页，然后单击Publishers按钮。在弹出的对话框中，选择“受信任的根证书颁发机构”标签页。如图17.3所示，你可以看到计算机信任的CA列表。

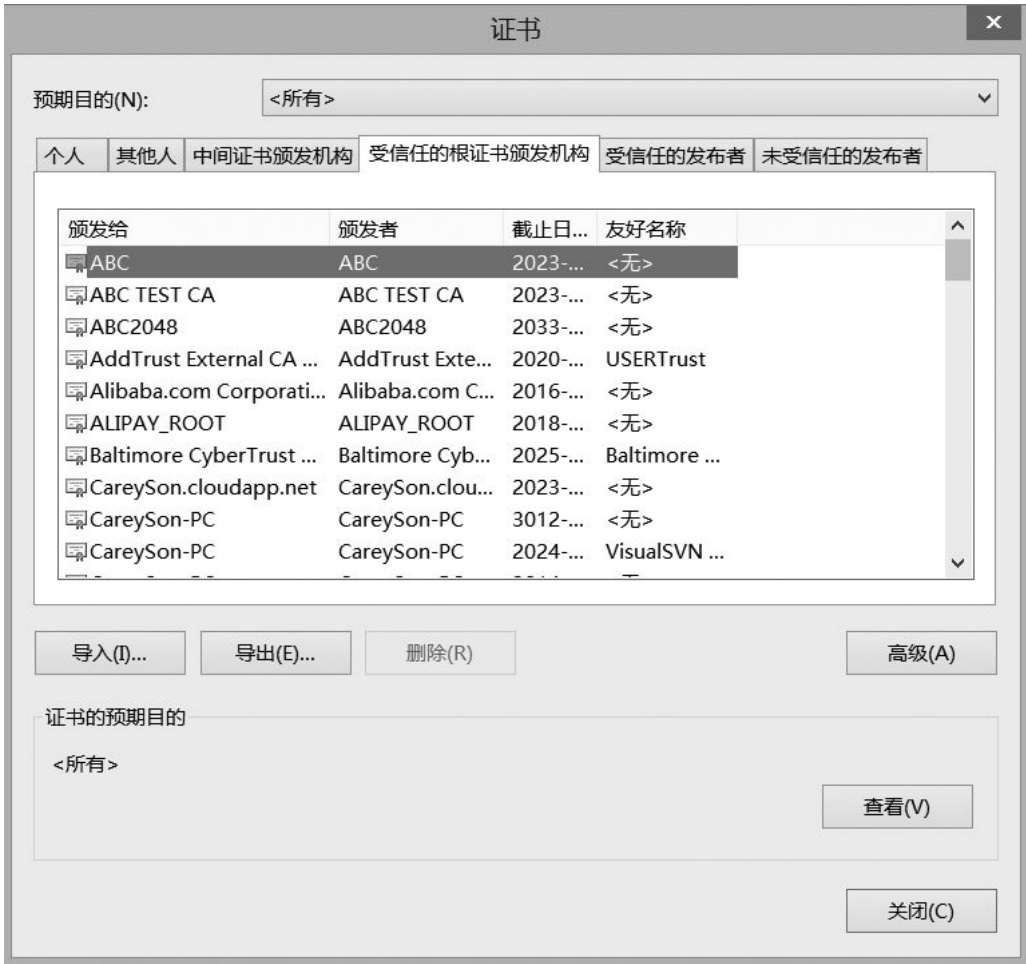


图17.3 设置计算机的“受信任的根证书颁发机构”选项

当你信任一个CA之后，你也会信任该CA签发的所有证书。如果有人使用一个证书对恶意脚本进行签名，那么你可以通过该证书去查找该脚本的作者——这也就是为什么已签名的脚本相对于未签名的脚本更加值得“信任”。但是如果你信任一个无法很好验证身份的CA，那么一个恶意脚本的作者可能会获取一个虚假的证书，这样你就无法使用该CA的证书去做追踪。这也就是为什么选择一个受信任的CA是如此重要。

一旦你获取了一个三级证书（具体而言，你需要一个包装为带有验证码的证书——通常CA会针对不同的操作系统以及不同的编程语言提供不同的证书），之后将该证书安装到本地计算机。安装之后，你可以使用

PowerShell的Set-AuthenticodeSignatureCmdlet将该数字签名应用到一段脚本。如果需要查看更详细的信息，你可以在PowerShell中执行Help About_Signing命令。许多商业的脚本开发环境（PrimalScript、PowerShell Plus以及PowerGUI等）都可进行签名，甚至可以在你保存一段脚本时进行自动签名，这样使得签名过程更加透明。

签名不仅会提供脚本作者的身份信息，也会确保在作者对脚本签名之后，不会被他人更改。实现原理如下：

（1）脚本作者持有一个数字证书，该密钥包含两个密钥：一个公钥、一个私钥。

（2）当对脚本进行签名时，该签名会被私钥加密。私钥仅能被脚本开发者访问，同时仅有公钥能对该脚本进行解密。在签名中会包含脚本的副本。

（3）当PowerShell运行该脚本时，它会使用作者的公钥（包含在签名中）解密该签名。如果解密失败，则说明签名被篡改，那么该脚本就无法被运行。如果签名中的脚本副本与明文文本不吻合，那么该签名就会被识别为损坏，该脚本也无法被运行。

图17.4描述了当执行脚本时，PowerShell处理的整个流程。在该流程中，你可以看到为什么AllSigned执行策略在某种意义上说更加安全：在该种执行策略下，仅有包含签名的脚本才能被运行，也就意味着，你总是能识别某段脚本的作者。如果需要执行某段脚本，那么就会要求对该脚本进行签名。当然，如果你修改了该脚本，你也就需要对该脚本重新签名（可能稍显烦琐）。

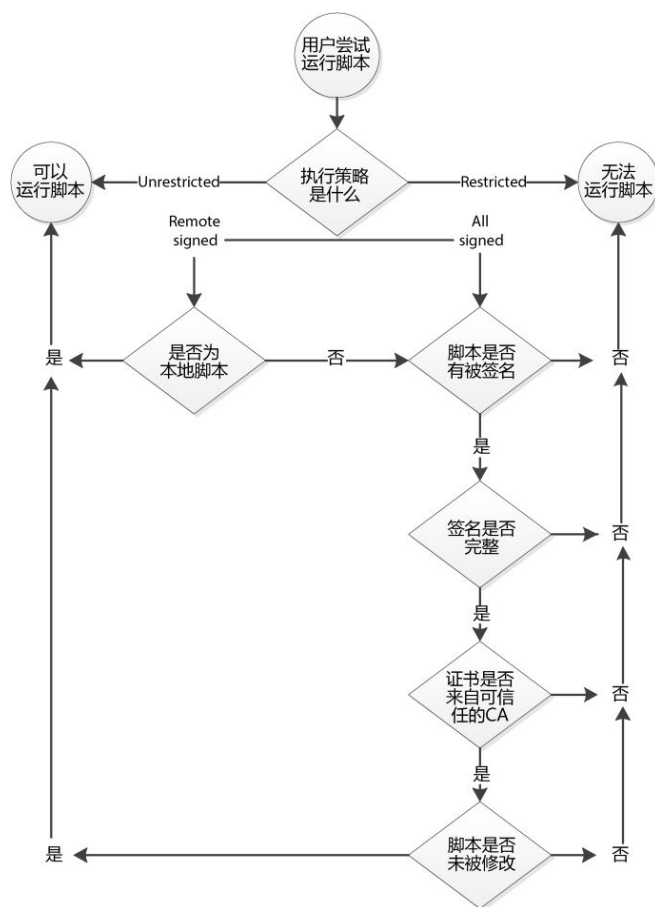


图17.4 尝试执行脚本时PowerShell的处理流程

17.4 其他安全措施

PowerShell包含另外两种总是一直有效的重要安全设置。一般情况下，它们应该保持默认值。

首先，Windows不会将PS1文件扩展名（PowerShell会将PS1识别为PowerShell的脚本）视为可执行文件类型。双击打开PS1文件，默认会使用记事本打开进行编辑，而不会被执行。该配置选项会保证用户不会在不知晓的情况下运行某段脚本，即使PowerShell的执行策略允许执行该脚本。

其次，在Shell中不能通过键入脚本名称去执行该脚本。Shell不会在当前目录中搜索脚本，也就是说，如果有一个名为test.PS1的脚本，切换到该脚本路径下，键入test或者test.PS1都不会运行该脚本。

比如下面的例子：

```
PS C:\> test
无法将“test”项识别为Cmdlet、函数、脚本文件或可运行程序的名称。请检查名称的拼写，如
果包括路径，请确保路径正确，然后重试。
所在位置行:1 字符:5
+ test<<<<
      + CategoryInfo          : ObjectNotFound: (test:String) [],
CommandNo
      tFoundException
      + FullyQualifiedErrorId :CommandNotFoundException

Suggestion [3,General]: 未找到命令 test，但它确实存在于当前位置。Windows
PowerSh
ell 默认情况下不从当前位置加载命令。如果信任此命令，请改为键入 ".\test"。有关更
多详细信息，请参阅 "get-help about_Command_Precedence"。
PS C:\>
```

通过上面的例子，你可以发现，PowerShell会检测该脚本，但是会给出警告信息：必须通过绝对路径或者相对路径来运行该脚本。因为test.PS1脚本位于C:目录下，所以你可以键入C:\test（绝对路径）或者运行.\test（指向当前路径的相对路径）。

该安全功能的目的是为了防止称为“命令劫持”的攻击类型。在该攻击中，它会将一个脚本文件放入到一个文件夹中，然后将它命名为某些内置的命令名，比如Dir。在PowerShell中，如果你在一个命令前面没有加上其路径——比如运行Dir命令，那么你很明确运行的这个命令的功能；但是如果运行的是.\Dir，那么你就会运行一个名为Dir.PS1的脚本。

17.5 其他安全漏洞

正如本章前面所讨论，PowerShell的安全主要在于防止用户在不知情的情况下运行不受信任的脚本。没有什么安全措施可以阻止用户向Shell手动键入命令或者拷贝一个脚本的全部内容，然后粘贴进Shell中（尽管以该种方式运行脚本，可能不会有相同的作用）。恶意脚本很难让用户去进行手动执行，以及指导用户如何去做，这也就是为什么微软并没有将该种场景作为一个潜在的攻击因素。但是请记住，PowerShell并不会给予用户额外的权限——用户仅能做权限允许的事情。

某些人可能会通过电话联系用户或者发送邮件方式，让用户打开PowerShell程序，然后键入一些命令，最后损坏他们的计算机。但是这些人也可以不通过PowerShell而是其他方式去攻击某些用户。说服一个用户打开资源管理器，选择到Program Files文件夹，然后按键盘上的Delete键是非常

容易的（当然，视你自己的真实情况，也可能比较困难）。在某些方面，比起让用户执行相同功能的PowerShell命令，这会更容易。

我们会指出这一点，是因为人们总是倾向于对命令行以及其看起来具备无限多的功能及功能延伸感到焦虑不安，但是事实上，你和你的用户如果通过其他方式无法完成某些工作，那么在PowerShell中你也是无法完成的。

17.6 安全建议

正如前面提到的，微软建议针对需要运行脚本的计算机，将PowerShell的执行策略设置为RemoteSigned。当然，你也可以考虑设置为AllSigned或者Unrestricted。

AllSigned选项相对来说可能比较麻烦，但是如果采用了下面两条建议，那么该选项会变得更加方便。

- 商业CA针对一个代码签名证书，每年最多收费900美金。如果你没有一个内部的PKI可以提供免费的证书，那么你也可以自己制作。运行Help About_Signing可以查询如何获取以及使用MakeCert.exe，该工具可以用来制作一个本地计算机信任的证书。如果你仅需在本地计算机运行脚本，那么这种方式是较快免费获取一个证书的方式。
- 通过我们上面提及的编辑器去编辑一段脚本，这些编辑器在你每次保存这些脚本时对脚本进行签名。通过这种方式，签名过程更加透明以及自动化，这样对用户来说更加方便。

正如前面所讲，我们都不太建议你去修改.PS1文件名的关联性。我们曾经看到过某些人修改了Windows的一些设置，将.PS1视为一种可执行文件，也就意味着，你可以通过双击一个脚本来执行它。如果采用这种方式，那么我们就回到使用VBScript时的糟糕日子，所以你需要避免该问题。

另外需要指出的是，我们在MoreLunches.com上提供的脚本都是没有经过数字签名的。也就意味着，这些脚本可能会在不知情的情况下被修改，最后脱离本意。所以在运行这些脚本之前，你应该花费一定的时间去检查它们，理解它们实现的功能，并且确保它们与本书中对应的脚本相吻合（如果可能的话）。我们之所以不对这些脚本进行签名，就是为了让你们花费这部分时间来完成这些工作：你应该养成这个习惯，对那些从网上下载脚本来检查，不管该脚本来自于多么受信任的作者。

17.7 动手实验

注意： 对于本次动手实验来说，你需要运行PowerShell v3或更新版本PowerShell的计算机。

在本章的动手实验中，你的任务非常简单——正因为如此简单，所以MoreLunches.com网站中没有给出参考答案。我们需要你通过一些配置选项使得PowerShell可以执行脚本。通过Set-ExecutionPolicyCmdlet，我们建议的值是RemoteSigned。当然，你也可以选择AllSigned这个值，但是对本书后面章节的动手实验环节来说可能就不太适合了。甚至你也可以选择Unrestricted执行策略。

即便如此，如果在生产环境中使用PowerShell工具，也请保证你选择的执行策略的设定值符合贵公司的安全规则与流程。我们不想你为了本书以及其动手实验而陷入某种困境。

第18章 变量：一个存放资料的地方

前面已经提到过，PowerShell包含脚本语言，并且在前面几章中已经开始与脚本语言打交道。但是一旦你开始使用脚本编程，就需要了解什么是“变量”，所以我们以此作为本章开端。你可以在其他复杂的脚本中使用变量，因此我们也会展示如何在这些地方使用变量。

18.1 变量简介

简单来说，变量就是在内存中的一个带有名字的“盒子”。你可以把所有你想存放的东西都放入这个“盒子”中：一个计算机名、一系列服务的集合、XML文档等。然后通过名字去访问这个盒子。在访问过程中，可以存放、添加或者从里面检索东西。这些东西是一直驻留在盒子里面的，并且允许你反复使用它们。

PowerShell并没有对变量有太多限制。比如，你不需要在使用变量前对其进行显式声明或定义。你也可以更改变量值的类型：某个时刻你可能只存储了一个单一进程在里面，下一时刻又可能存储一系列的计算机名进去。变量甚至可以存储多种不同的东西，比如服务的集合和进程的集合（虽然允许这样做，但是大部分情况下，使用变量的内容还是有讲究的）。

18.2 存储值到变量中

PowerShell中的所有东西——的确是所有东西，都被认为是一个对象。即使一个简单的字符串，比如计算机名，都被当作对象对待。比如，把一个字符串用管道传输到Get-Member（或者它的别名Gm），可以看到对象的类型是“System.String”，并且有很多方法可用（为了节省空间，这里截断了部分输出）。

```
PS C:\> "SERVER-R2" | gm  
  
TypeName: System.String
```

Name	MemberType	Definition
Clone	Method	System.Object Clone()
CompareTo	Method	int CompareTo(System.Object valu...
Contains	Method	bool Contains(string value)
CopyTo	Method	System.Void CopyTo(int sourceInd...
EndsWith	Method	bool EndsWith(string value), boo...
Equals	Method	bool Equals(System.Object obj), ...
GetEnumerator	Method	System.CharEnumerator
GetEnumera...		
GetHashCode	Method	int GetHashCode()
GetType	Method	type GetType()
GetTypeCode	Method	System.TypeCode GetTypeCode()
IndexOf	Method	int IndexOf(char value), int
Ind...		
IndexOfAny	Method	int IndexOfAny(char[] anyOf), in...

动手实验： 在你自己的电脑上运行命令，看是否能获取来自于“System.String”对象的完整的方法和属性的列表。

虽然技术上字符串是一个对象，但是和其他Shell中的东西一样，你会发现人们更倾向于把它当作一个简单的值。因为大部分情况下，我们关注的是它的值（如前面提到的“SERVER-R2”），而不会过多关注从属性中查找信息。也就是说，一个进程就算很庞大，数据结构很抽象，而你通常只需要处理一些单独的属性，如VM、PM、Name、CPU、ID等。一个字符串是一个对象，但是比常见的进程，它又显得没那么复杂。

PowerShell允许在一个变量中存储简单的值。你需要定义一个变量，然后使用等号符（=），用于赋值操作，接下来是变量所需存储的值。下面是例子。

```
PS C:\> $var = "SERVER-R2"
```

动手实验： 希望你能动手运行这些例子，以便你能重现我们的结果。但是需要把服务器名改为本地，而不是使用“SERVER-R2”。

需要注意的是，美元符（\$）并不是变量名的一部分。在我们的例子中，变量名是“var”。“\$”符只是告知Shell接下来的是一个变量名，并且将要赋值给这个变量。

- 下面我们看看关于变量及其名字的一些注意事项。
- 变量名通常包含字母、数字和下划线，最常见的形式是以字母或下划线开头。
- 变量名可以包含空格，但是名字必须被花括号包住。比如\${My Variable}，表示一个变量名“My Variable”。就我个人而言，我不喜欢变量名包含空格，因为这会要求更多的输入操作，并且不易阅读。
- 变量不驻留在Shell会话之间。当关闭Shell时，所有你创建的变量都会清除。
- 变量名可以很长——长到你可以不用考虑它到底能有多长。但是请确保变量名的可读性。比如，如果你想要把计算机名存入变量，可以使用“computername”作为变量名。如果变量需要包含一系列的进程，使用“processes”是个不错的选择。
- 除了有VBScript背景的人，PowerShell用户通常不需要使用前缀名来标识变量存放了什么。比如在VBScript中，“strComputerName”是常见的变量名，表示变量存储的是一个字符串（“str”部分）。PowerShell不在意你是否这样做。同时在大多数社区中，这种习惯也不认为是好习惯。

如果需要查询变量的内容，可以使用美元符号加上变量名，像下面的例子来实现。再次提醒，美元符只是告诉Shell你需要访问变量内容；紧跟其后的变量名才是告诉Shell你要访问的变量是什么。

```
PS C:\> $var  
SERVER-R2
```

你可以在几乎所有地方使用变量来替代值。比如，当使用WMI时，你可以选择指定一个计算机名。这个命令类似：

```
PS C:\> get-wmiobject win32_computersystem -comp SERVER-R2  
  
Domain           : company.pri  
Manufacturer     : VMware, Inc.  
Model            : VMware Virtual Platform  
Name             : SERVER-R2
```

```
PrimaryOwnerName      : Windows User
TotalPhysicalMemory   : 3220758528
```

然后可以使用变量来替代这个值:

```
PS C:\> get-wmiobject win32_computersystem -comp $var

Domain                : company.pri
Manufacturer          : VMware, Inc.
Model                 : VMware Virtual Platform
Name                  : SERVER-R2
PrimaryOwnerName      : Windows User
TotalPhysicalMemory   : 3220758528
```

顺带说说, **var**的确是我们常见的变量名。我们认为使用“**computername**”是不错的选择, 但是在一些特殊地方, 将会重复使用**\$var**作为变量名, 所以这里还是保持使用**var**。但不要因为这个例子使你放弃使用有意义的名字作为变量名。

下面将从赋值给**\$var**开始, 但是会在任何时候更改它的值。

```
PS C:\> $var = 5
PS C:\> $var | gm
    TypeName: System.Int32

Name      MemberType Definition
-----
CompareTo Method      int CompareTo(System.Object value), int
CompareT...
Equals    Method      bool Equals(System.Object obj), bool
Equals(int ...
GetHashCode Method     int GetHashCode()
GetType   Method     type GetType()
GetTypeCode Method    System.TypeCode GetTypeCode()
ToString  Method     string ToString(), string ToString(string
format...
```

在前面的例子中, 我们把一个数值放入**\$var**, 然后把**\$var**与**Gm**用管道相连接。可以看到, **Shell**把**\$var**的内容识别成**System.Int32**, 或一个32位数值。

18.3 使用变量：有趣的引号

前面我们一直在讨论变量，是时候覆盖一个完整的PowerShell特性。关于这一点，我们已经在书中建议过，使用单引号包住字符串。因为PowerShell会把所有包在单引号中的东西认为是一个文本字符串。如下面的例子：

```
PS C:\> $var = 'What does $var contain?'
PS C:\> $var
What does $var contain?
```

在前面的例子中可以看到，在单引号包含部分中的\$var被认为是一个文本字符。

但是在双引号中又是另外一番情景。看看下面的技巧：

```
PS C:\> $computername = 'SERVER-R2'
PS C:\> $phrase = "The computer name is $computername"
PS C:\> $phrase
The computer name is SERVER-R2
```

我们首先把“SERVER-R2”存入变量“\$computername”。然后在变量\$phrase中存储““The computer name is \$computername””，这里使用的是双引号。PowerShell自动在双引号中搜索美元符，然后变量的值替代所有被找到的变量。因为这里展示的是\$phrase的内容，所以\$computername变量被“SERVER-R2”替代。

这种替代操作仅发生在Shell初次解析字符串的时候。此时，\$phrase包含的是“The computer name is SERVER-R2”——它并没有包含“\$computername”字符串。可以通过修改\$computername的内容检查\$phrase是否自己更新：

```
PS C:\> $computername = 'SERVER1'
PS C:\> $phrase
The computer name is SERVER-R2
```

可以看到，`$phrase`变量依旧保存原有的值。

关于PowerShell双引号的另外一个窍门是转义字符。这个字符是重音符（```），在美式键盘左上角的部分，通常在Esc键的下方，与波浪符（`~`）在同一个键上。使用重音符的问题是，在某些字体中，很难区分单引号。实际上，我们常常使用Consolas字体，因为它较Lucida Console 或 Raster字体更容易区分重音符。

动手实验：单击PowerShell窗口左上角的控件，选择属性。在【字体】标签页，选择如图18.1所示的Consolas 字体，再单击【OK】按钮。然后输入一个单引号和重音符看是否能区分它们。图18.1 显示了在我们系统中的样子。你能从中看出区别吗？我相信，使用足够大的字体时是可以的。区分起来有点困难，所以请你选择合适的字体和大小，以便你可以轻易地区分出它们。



图18.1 设置字体以便更容易区分单引号和重音符

下面来看看转义字符的作用。它消除了与它关联后的有特殊意义的字符，或在某些情况下增加了字符的特殊意义。下面使用前面用过的例子：

```
PS C:\> $computername = 'SERVER-R2'
PS C:\> $phrase = "`$computername contains $computername"
PS C:\> $phrase
$computername contains SERVER-R2
```

当我们把字符串赋给\$phrase时，我们使用了两次\$computername。第一次，我们在美元符前使用了重音符。这样去除了美元符在变量符号中的特殊意义，并把它当作字符中的美元符号。从前面的输出中可以看出，在最后一行，\$computername是存储在变量中的。在第二次时，没有使用重音符，所以\$computername被变量值替换掉。

下面来看一个第二种使用重音符的例子。

```
PS C:\> $phrase = "`$computername`ncontains`n$computername"
PS C:\> $phrase
$computername
contains
SERVER-R2
```

仔细检查，你会发现我们在语句中使用了两次`n——一个在第一个\$computername后，另外一个在contains后。在这个例子中，重音符作为添加特殊功能而存在。一般来说，“n”是一个字母，但是在前面带有重音符之后，它就变成了一个回车与换行符（n是new line的意思）。

运行“help about_escape”可以获取更多的信息，它包含了其他关于特殊转义符的列表。你可以尝试使用转义后的“t”来实现tab功能，或者使用转义后的“a”来使机器发出响声（a是alert，警报的意思）。

18.4 存储多个对象在一个变量中

在此之前，我们都是针对单一对象来介绍变量，并且这些变量都是简单的值。我们都是直接操作这些对象本身而不是它们的属性或者方法。现在我们尝试把一堆对象放入一个单一变量中。

其中一种方式是使用逗号分隔符列表，因为PowerShell认为这些列表是对象的集合。

```
PS C:\> $computers = 'SERVER-R2','SERVER1','localhost'
PS C:\> $computers
SERVER-R2
SERVER1
localhost
```

请留心观察上面的例子，逗号是放在单引号外面的。如果把这些逗号放在单引号里面，会变成一个包含逗号和三个计算机名的单一对象。通过我们的方法，可以得到三个独立的对象，它们的类型均为字符串类型。正如你所看到的，当我们检查变量的内容时，PowerShell会把每个对象分别以单行展示。

18.4.1 与多值单一变量的单一对象交互

你可以在某一时刻访问多值单一变量（一个变量存储多个值）的独立元素，只需在中括号中指定你要访问的对象的索引号即可。这个号从0开始，第二个值的索引号为1，以此类推。你还可以使用 - 1这个索引号来访问对象的最后一个值，- 2为倒数第二个值，等等。比如：

```
PS C:\> $computers[0]
SERVER-R2
PS C:\> $computers[1]
SERVER1
PS C:\> $computers[-1]
localhost
PS C:\> $computers[-2]
SERVER1
```

变量本身有一个属性可以查看其中包含多少个对象：

```
PS C:\> $computers.count
3
```

你同样可以访问变量内部对象的属性和方法，就像变量自身的属性和方法一样。首先，针对只有单一对象的变量：

```
PS C:\> $computername.length
9
PS C:\> $computername.toupper()
SERVER-R2
PS C:\> $computername.tolower()
server-r2
PS C:\> $computername.replace('R2','2008')
SERVER-2008
PS C:\> $computername
SERVER-R2
```

在前面的例子中，我们使用了本章前面创建的变量`$computername`。你是否还记得这个变量包含了一个类型为`System.String`的对象，并且在18.2节中已经通过与`Gm`进行管道传输后得到关于这个类型的属性和方法的完整列表。在这里，我们使用了`Length`、`ToUpper()`、`ToLower()`和`Replace()`方法。在每一个例子中，即使`ToUpper()`和`ToLower()`都不要要求括号中出现任何值，但是我们也要在方法名后使用括号。同时可以看到这些方法都没有修改变量中的任何事物——可以看结果的最后一行。取而代之的是，每个方法都在原有基础上创建了一个新的字符串结果，正如由方法修改过一样。

18.4.2 与多值单一变量的多个对象交互

当一个变量包含了多个对象，处理步骤变得稍微有点麻烦。即使变量中的每个对象都具有相同的类型，比如前面例子中的`$computers`变量，但是PowerShell v2并不允许你同时针对多个对象调用一个方法或者访问一个属性。如果你非要尝试，会收到报错信息。

```
PS C:\> $computers.toupper()
Method invocation failed because [System.Object[]] doesn't contain a
metho
d named 'toupper'.
At line:1 char:19
+ $computers.toupper <<<< ()
    + CategoryInfo          : InvalidOperation: (toupper:String) [],
Run
imeException
```

```
+ FullyQualifiedErrorId : MethodNotFound
```

取而代之的是，你必须指定变量中你想操作的那个对象，然后访问它的属性或执行一个方法。

```
PS C:\> $computers[0].tolower()  
server-r2  
PS C:\> $computers[1].replace('SERVER','CLIENT')  
CLIENT1
```

再次提醒，这些方法会产生新的字符串结果，而不会更改变量中的那些原有值。用下面的方式可以测试。

```
PS C:\> $computers  
SERVER-R2  
SERVER1  
Localhost
```

如果你希望修改变量中的内容，该怎么办呢？你必须在现有的其中一个对象中赋予新值。

```
PS C:\> $computers[1] = $computers[1].replace('SERVER','CLIENT')  
PS C:\> $computers  
SERVER-R2  
CLIENT1  
Localhost
```

从例子中可以看出已经修改了变量里面的第二个对象，而不是产生一个新的字符串。我们在这里提出的这个例子仅在安装了PowerShell v2的电脑上才有效；而这种行为已经在v3中得到改变，我们将会在后面介绍。

18.4.3 与多个对象交互的其他方式

我们将会介绍在包含多个对象的单个变量中与它们的属性和方法交互的两种选项。在前面的例子中，仅仅执行了变量中单个对象的方法。

如果你想要变量中的每个对象都执行**ToLower()**方法，并把结果存储回去，你可以像这样执行：

```
PS C:\> $computers = $computers | ForEach-Object { $_.ToLower() }
PS C:\> $computers
server-r2
client1
localhost
```

这个例子稍微有些复杂，所以我们在图18.2中把它分解。首先，**\$computers =**与管道相连，意味着管道的输出将会被存储在变量中。这些结果将会覆盖以前变量的所有值。

管道从**\$computers**开始，并传输到“**ForEach-Object**”。这个Cmdlet会枚举管道中的所有对象（这里总共有3个计算机名并且是字符串对象），然后执行对应的代码块。在每个代码块中，**\$_**占位符每次都包含一个被管道传输进来的对象，然后针对每个对象执行**ToLower()**方法。最后由**ToLower()**产生的字符串对象会被放入管道——然后存入**\$computers**变量。

你可以使用“**Select-Object**”来操作类似的属性。例子中是查询每个用于管道连接的对象的**Length**属性：

```
PS C:\> $computers | select-object length
Length
-----
9
7
9
```

因为属性是数值型，所以**PowerShell**把输出以右对齐的方式展示。

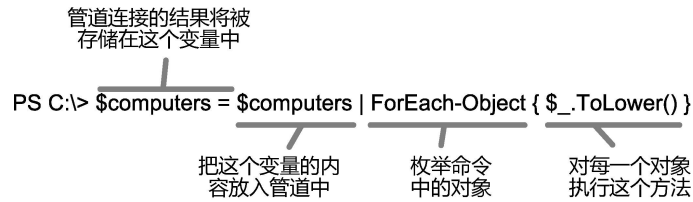


图18.2 使用“ForEach-Object”方法执行在变量中包含的每个对象上

18.4.4 在PowerShell中展现属性和方法

“当一个变量包含多个对象时，不能访问属性和方法”被证明是会让 PowerShell v1和v2 用户非常头痛的要求。因此，对于v3和后续版本，微软做出重要改变，名为“automatic unrolling”。它本质上意味着你现在可以访问一个包含多个对象的单个变量的属性和方法：

```
$services = Get-Service
$services.Name
```

底层实现中，PowerShell会意识到你正在尝试访问一个属性。同样，它也知道在`$services`集合中没有一个关于名称的属性——但是集合中的独立对象有这个属性。所以它隐式枚举，或展现对象，并获取每个对象的名称属性。这等于：

```
Get-Service | ForEach-Object { Write-Output $_.Name }
```

也可以等于：

```
Get-Service | Select-Object -ExpandProperty Name
```

这两种方式是在v1 和v2中不得不用方式，其工作原理也等于：

```
$objects = Get-WmiObject -class Win32_Service -filter "name='BITS'"
$objects.ChangeStartMode('Disabled')
```

记住，这是在PowerShell v3和后续特性中独有的——不要期望这种方式能在旧版本中有效。

18.5 双引号的其他技巧

对于双引号，还有一个很酷的技术可用，这个技巧是对变量替换的概念延伸。假设你把一堆服务存入`$service`变量。现在你只想把第一个服务名放入一个字符串：

```
PS C:\> $services = get-service
PS C:\> $firstname = "$services[0].name"
PS C:\> $firstname
AeLookupSvc ALG AllUserInstallAgent AppIDSvc Appinfo AppMgmt
AudioEndpoint
Builder Audiosrv AxInstSV BDESVC BFE BITS BrokerInfrastructure
Browser bth
serv CertPropSvc COMSysApp CryptSvc CscService DcomLaunch defragsvc
Device
AssociationService DeviceInstall Dhcp Dnscache dot3svc DPS DsmSvc
Eaphost
EFS ehRecvr ehSched EventLog EventSystem Fax fdPHost FDResPub fhsvc
FontCa
che gpsvc hidserv hkmsvc HomeGroupListener HomeGroupProvider IKEEXT
iphlp
vc KeyIso KtmRm LanmanServer LanmanWorkstation lltdsvc lmhosts LSM
Mcx2Svc
MMCSS MpsSvc MSDTC MSiSCSI msiserver napagent NcaSvc NcdAutoSetup
Netlogo
n Netman netprofm NetTcpPortSharing NlaSvc nsi p2pimsvc p2psvc
Parallels C
herence Service Parallels Tools Service PcaSvc PeerDistSvc PerfHost
pla P
lugPlay PNRPAutoReg PNRPsvc PolicyAgent Power PrintNotify ProfSvc
QWAVE Ra
sAuto RasMan RemoteAccess RemoteRegistry RpcEptMapper RpcLocator
RpcSs Sam
Ss SCardSvr Schedule SCPolicySvc SDRSVC seclogon SENS SensrSvc
SessionEnv
SharedAccess ShellHWDetection SNMPTRAP Spooler sppsvc SSDPSRV SstpSvc
stis
vc StorSvc svsvc swprv SysMain SystemEventsBroker TabletInputService
TapiS
rv TermService Themes THREADORDER TimeBroker TrkWks TrustedInstaller
UI0De
tect UmRdpService upnphost VaultSvc vds vmicheartbeat vmickvpexchange
vmic
rdv vmicshutdown vmictimesync vmicvss VSS W32Time wbengine WbioSvc
```

```
Wcmsvc
  wcnscsv WcsPlugInService WdiServiceHost WdiSystemHost WdNisSvc
WebClient
Wecsv wercplsupport WerSvc WiaRpc WinDefend WinHttpAutoProxySvc
Winmgmt W
inRM WlanSvc wlidsvc wmiApSrv WMPNetworkSvc WPCSvc WPDBusEnum wscsv
WSear
ch WSService wuauserv wudfsvc WwanSvc[0].name
```

惨了，出错了。例子中紧跟`$services`的`[`符不是常规文本字符，会引发PowerShell尝试替换`$services`。同时因为这种阻塞，字符串中的`[0].name`部分完全没有被替换。

解决方法是这些所有东西放入一个表达式：

```
PS C:\> $services = get-service
PS C:\> $firstname = "The first name is $($services[0].name)"
PS C:\> $firstname
The first name is AeLookupSvc
```

在`$()`中的所有东西会被当成普通的PowerShell命令，结果也被放入字符串中，替代原有的所有东西。同样，这种操作仅在双引号中有效。这种`$()`结构称为子表达式。

另外，在PowerShell v3及后续版本中还有一个很酷的功能。有时候，你需要把更复杂的内容放入一个变量，然后在引号中显示变量的内容。在PowerShell v3及后续版本中，Shell能更智能地枚举集合中的所有对象。即使你仅引用一个属性或方法，作用域集合中所有相同类型的对象中也没问题。比如，我们查询服务的清单并把它们放入`$service`变量中，然后使用双引号仅包含服务的名称：

```
PS C:\> $services = get-service
PS C:\> $var = "Service names are $services.name"
PS C:\> $var
Service names are AeLookupSvc ALG AllUserInstallAgent AppIDSvc
Appinfo App
Mgmt AudioEndpointBuilder Audiosrv AxInstSV BDESVC BFE BITS
BrokerInfratr
ucture Browser bthserv CertPropSvc COMSysApp CryptSvc CscService
DcomLaunc
h defragSvc DeviceAssociationService DeviceInstall Dhcp Dnscache
```



```
dot3svc D
PS DsmSvc Eaphost EFS ehRecvr ehSched EventLog EventSystem Fax
fdPHost FDR
esPub fhsvc FontCache FontCache3.0.0.0 gpsvc hidserv hkmsvc
HomeGroupListe
ner HomeGroupProvider IKEEXT iphlpsvc KeyIso KtmRm LanmanServer
LanmanWork
station lltdsvc lmhosts LSM Mcx2Svc MMCSS MpsSvc MSDTC MSiSCSI
msiserver M
SSQL$SQLEXPRESS napagent NcaSvc NcdAutoSetup Netlogon Netman netprofm
NetT
cpPortSharing NlaSvc nsi p2pimsvc p2psvc Parallels Coherence Service
Paral
l els Tools Service PcaSvc PeerDistSvc PerfHost pla PlugPlay
PNRPAutoReg PN
RPsvc PolicyAgent Power PrintNotify ProfSvc QWAVE RasAuto RasMan
RemoteAcc
ess RemoteRegistry RpcEptMapper RpcLocator RpcSs SamSs SCardSvr
Schedule S
CPolicySvc SDRSVC seclogon SENS SensrSvc SessionEnv SharedAccess
ShellHWDe
```

这里截断了一部分输出以便节省空间，但是我们希望你能理解这种思想。显然，这些可能并不是你希望查询的结果。但是从前面提到的子表达式和这里的例子中，你应该能得到一些启示。

18.6 声明变量类型

目前为止，我们仅仅把对象存入变量并让PowerShell指出我们正在使用的对象的类型。这是因为PowerShell不在乎你放入变量中的对象是什么类型，但是我们在意。

比如，假设你有一个变量希望用于存储一个数值，准备用于一些算术运算，并期待用户输入一个数值。请看下面的例子，你可以直接在命令行中输入数值：

```
PS C:\> $number = Read-Host "Enter a number"
Enter a number: 100
PS C:\> $number = $number * 10
PS C:\> $number
100100100100100100100100100100
```

动手实验： 目前为止，我们没有提到“Read-Host”——我们将把它放到下一章介绍——但是如果你跟着做实验，它的功能还是很明显的。

见鬼，为什么100乘以10会得出100100100100100100100100100100？这是什么数字？

如果你眼尖，你可以发现，PowerShell并没有把我们的输入当作数值，而是把它当作字符串。PowerShell只是把100这个字符串重复了10次，而不是把100乘以10。所以结果就是把字符串100在一行中列了10次。

我们可以用下面的方式验证。

```
PS C:\> $number = Read-Host "Enter a number"
Enter a number: 100
PS C:\> $number | gm

    TypeName: System.String

Name      MemberType      Definition
----      -
Clone      Method           System.Object Clone()
CompareTo   Method           int CompareTo(System.Object
valu...
Contains    Method           bool Contains(string value)
```

通过把\$number用管道传输到Gm中，可以看出Shell把它视为System.String，而不是System.Int32。对于这个问题有很多解决方法，我们将介绍其中最简单的一种。

首先，告诉Shell知道\$number变量应该存储一个整型，强制Shell把值转换成一个实数。如下面的例子，通过在变量首次使用前使用[]，明确定义一个数据类型“int”来实现：

```

PS C:\> [int]$number = Read-Host "Enter a number"
Enter a number: 100
PS C:\> $number | gm

```

强制类型转换
① 换成[int]

```

    TypeName: System.Int32

```

Name	MemberType	Definition
CompareTo	Method	int CompareTo(System.Object value), int CompareT...
Equals	Method	bool Equals(System.Object obj), bool Equals(int ...
GetHashCode	Method	int GetHashCode()
GetType	Method	type GetType()
GetTypeCode	Method	System.TypeCode GetTypeCode()
ToString	Method	string ToString(), string ToString(string format...

确认变量的数据
② 类型是 Int32

```

PS C:\> $number = $number * 10
PS C:\> $number
1000

```

变量被处理成
③ 数值型

在前面的例子中，我们使用了[int]强制\$number仅包含整数①。在你输入以后，我们把\$number用管道传输到Gm，验证它的确已经是整型而不是字符串②。最后我们可以看到，变量的值被认为是数值型并进行了实际乘法运算③。

这个技术的另外一个强项是，在Shell不能把数据的值转换成数字时，让Shell可以抛出错误，因为\$number仅仅是存储数值的一个容器。

```

PS C:\> [int]$number = Read-Host "Enter a number"
Enter a number: Hello
Cannot convert value "Hello" to type "System.Int32". Error: "Input
string
was not in a correct format."
At line:1 char:13
+ [int]$number <<<< = Read-Host "Enter a number"
    + CategoryInfo          : MetadataError: (:) [],
ArgumentTransformationException
    + FullyQualifiedErrorId : RuntimeException

```

这是一个防止后续问题的例子，因为你可以确保\$number能存储你希望的值。

除了[int]之外，还有很多其他的选择。下面是最常用的一些类型清单。

- [int]——整型数字。

- [single]和[double]——单精度和多精度浮点型数值（小数位部分的数值）。
- [string]——字符串。
- [char]——仅单个字符（如[char]\$c='X'）。
- [xml]——一个XML文档。不管你是否解析里面的值，都要确保它包含有效的XML标记（比如[xml]\$doc=Get-Content MyXML.xml）。
- [adsi]——一个活动目录服务接口（ADSI）查询。Shell会执行查询并把结果对象存入变量（如[adsi]\$user="WinNT:\MYDOMAIN\Administrator,user"）。

明确指定变量的对象类型，可以避免在复杂脚本中出现一些严重的逻辑错误。正如下面的例子所示，一旦你指定了对象类型，PowerShell会强制它使用这种类型，直到重新显式定义变量的类型。

```
PS C:\> [int]$x = 5
PS C:\> $x = 'Hello'
Cannot convert value "Hello" to type "System.Int32". Error: "Input string
was not in a correct format."
At line:1 char:3
+ $x <<<< = 'Hello'
    + CategoryInfo          : MetadataError: (:) [], ArgumentTransformati
onMetadataException
    + FullyQualifiedErrorId : RuntimeException

PS C:\> [string]$x = 'Hello'
PS C:\> $x | gm

TypeName: System.String
```

1 定义变量\$x 为整型。

2 创建一个错误，并把错误信息放到\$x 中。

3 以字符形式重新对\$x 赋值。

4 确认\$x 的新类型。

Name	MemberType	Definition
Clone	Method	System.Object Clone()
CompareTo	Method	int CompareTo(System.Object valu...

在前面的例子中，你可以看到，我们首先声明\$x变量作为整型①，并把一个整型值放入变量。当我们准备把一个字符串放入变量时②，PowerShell抛出错误，因为它不能把字符串转换成整型数值。在后续把变量类型重新声明为字符串后，就可以把字符串放入其中③。通过管道把变量传输到Gm，可以查看变量的类型名④。

18.7 与变量相关的命令

我们虽然使用了变量，但是目前为止还没有正式地表明我们的意图。PowerShell不建议使用高级的变量声明，并且你不能强制声明。（试图去搜寻类似Option Explicit的VBScript使用者可能会感到沮丧，

PowerShell有类似的Set-StrictMode，但是并不完全一样。）但是Shell却包含了下面与变量有关的命令。

- New-Variable;
- Set-Variable;
- Remove-Variable;
- Get-Variable;
- Clear-Variable。

除了“Remove-Variable”之外，其他命令可能都不会用上。这个命令对需要删除的变量很有用（你也可以在变量中使用Del命令，驱动其删除这个变量）。你可以使用其他功能——创建新的变量、读取变量和配置变量——如使用本章提到过的即席语法（ad hoc syntax）；在大部分情况下，使用这些Cmdlets并没有带来什么特殊的优点。

如果你真的决定使用这些Cmdlets，需要把变量名授予对应Cmdlets的-name参数。这里仅需要变量名——不需要包含美元符。通常只有在操作类似超出作用域（out-of-scope）变量时，才可能用到这些Cmdlets。使用这种变量是很不好的习惯，所以本书不打算讲述这类变量，但是可以使用“help about_scope”来获取更详细的信息。

18.8 针对变量的最佳实践

虽然我们前面已经提到过绝大部分的最佳实践，但是还是有必要做一个快速回顾：

- 确保变量名有意义，但也要简洁。比如\$computername是一个很好的变量名，因为它清晰简短，\$c就不是，因为它不具有什么实际意义。变量名\$computer_to_query_for_data略微长了点儿。虽然它也有意义，但是我希望反反复复地输入它吗？
- 不要在变量中使用空格。虽然你可以这样做，但是这种语法相当不好。
- 如果变量仅包含一类对象，那么在你首次使用变量时，请定义对象类型。这样可以帮助你避免一些常见的逻辑错误，并且当你在商用脚本开发环境中工作时（PrimalScript也许就是其中一个例子），编辑软件可以在你告诉它变量将包含的对象类型时提供一些提示功能。

18.9 常见误区

对于初学者来说，最常见的误区是变量名。我希望在这一章中已经说得很清楚，但是请记住，美元符并不是变量包含的部分。它只是让Shell知道你想访问变量的内容，而美元符后面的才是变量名本身。

Shell有两个解析规则用于获取变量名：

- 如果紧随美元符后的字符是一个字母、数字或下划线，则变量名包含美元符到下一个空白的所有字符（可能是一个空格、Tab或回车）。
- 如果紧随美元符后的是一个左花括弧，则变量名包含左花括号开始但不包含右花括号之间的所有内容。

18.10 动手实验

注意： 对于本次动手实验来说，你需要运行PowerShell v3或更新版本PowerShell的计算机。

回到第15章，释放后台作业的内存，然后在命令行中执行下面的操作。

1. 创建一个后台作业，从两台计算机中查询Win32_BIOS信息（如果你只有一台计算机做实验，可以使用两次“localhost”来模拟）。
2. 当作业运行完毕后，把作业的结果存入一个变量。
3. 显示变量的内容。
4. 把变量内容导出到一个CliXML文件中。

18.11 进一步学习

花点时间浏览一下本书的前面章节。设计变量的目的是存储一些你可能需要反复使用的东西。你可以在前面章节中找到变量的用处吗？

比如，在第13章中，你已经学到创建一个远程计算机的链接。你在本章中学到的是如何在一个步骤中创建、使用和关闭链接。它不正是在几个命令中创建连接，存入到变量中吗？那只是其中一个用上变量的例子（我们将在第20章介绍）。看看你能否找到更多的例子。

第19章 输入和输出

到现在为止，在本书中，我们主要依赖PowerShell源生的能力来输出表格和列表。当你开始将多个命令整合成更复杂的脚本时，你可能想要更精确地控制展示的结果。你可能也希望能提示用户进行输入。在本章中，你将会学习到如何收集输入以及如何展示期望的输出结果。

19.1 提示并显示信息

PowerShell如何展示信息和进行对应的提示，依赖于PowerShell运行的方式。你可以看到，PowerShell被内置为一种底层的引擎。

与你进行交互的对象称为主机应用程序。当运行PowerShell.exe时，你看到的命令行控制台称为控制台主机。图形化的PowerShell ISE通常被称为ISE主机或者图形化主机。其他非微软的应用程序也可以调用PowerShell的引擎。你与主机应用程序进行交互，之后主机应用程序将执行的命令传递给该引擎。主机应用程序会展现引擎产生的结果集。

图19.1说明了PowerShell引擎和多种主机应用程序之间的关系。每个主机应用程序负责图形化展现引擎产生的任何输出结果，同时负责通过界面收集引擎需要的任何输入信息。也就意味着，PowerShell可以通过多种方式展现执行结果和收集输入信息。实际上，控制台主机和ISE使用不同的方法来收集输入信息：控制台主机会在命令行中展现一个文本的提示框，但是ISE会弹出一个会话框，该会话框中包含文本区域一个“OK”按钮。

我们希望指出这些差异点，因为有些时候可能会使得初学者非常困惑。为什么一个命令在命令行中的行为与在ISE中的行为大相径庭？这是因为你与Shell交互的方式由主机应用程序决定，并不是由PowerShell本身决定。我们即将展示给你的命令会显示使用不同的行为，这些行为主要依赖于你在哪里执行这些命令。

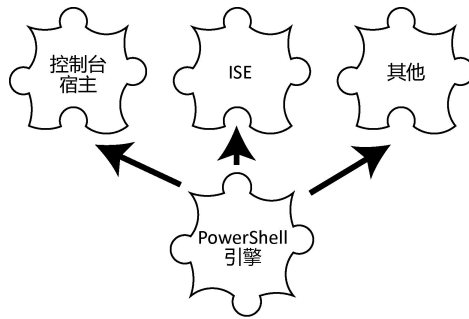


图19.1 多种应用程序都可以使用PowerShell引擎

19.2 Read-Host命令

PowerShell的Read-Host Cmdlet的功能是展示一个文本提示框，然后收集来自用户的输入信息。因为在前一章节中，你第一次看到我们使用这个Cmdlet，所以你会觉得语法比较熟悉：

```
PS C:\> Read-Host "Enter a computer name"
Enter a computer name: SERVER-R2
SERVER-R2
```

该示例突出了Cmdlet的两个重要的事实：

- 提示信息最后添加了一个冒号。
- 用户键入的任何信息都会作为该Cmdlet的返回结果（严格来说，键入的信息被放进了管道）。

你经常会将该输入信息传递给一个变量，类似下面这样：

```
PS C:\> $ComputerName=Read-Host "Enter a computer name"
Enter a computer name: SERVER-R2
```

动手实验： 现在请开始跟着这些示例学习吧。此时，`$ComputerName`变量中应该存在一个有效的计算机名称。除非使用的计算机名称是Server-2，否则请不要使用Server-2。

正如前面提到的，第二版的PowerShell ISE会展现一个对话框，而不是直接在命令行中进行提示，如图19.2所示。其他的主机应用程序，比如PowerGUI、PowerShell Plus或者PrimalScript等脚本编辑器，均使用各自对应的方式执行Read-Host。请记住，第三版的PowerShell ISE仅会展示更简单的两个窗格。不像第二版的PowerShell ISE那样，第三版的PowerShell ISE会像常规的控制台窗口一样展示一个命令行的提示窗口。

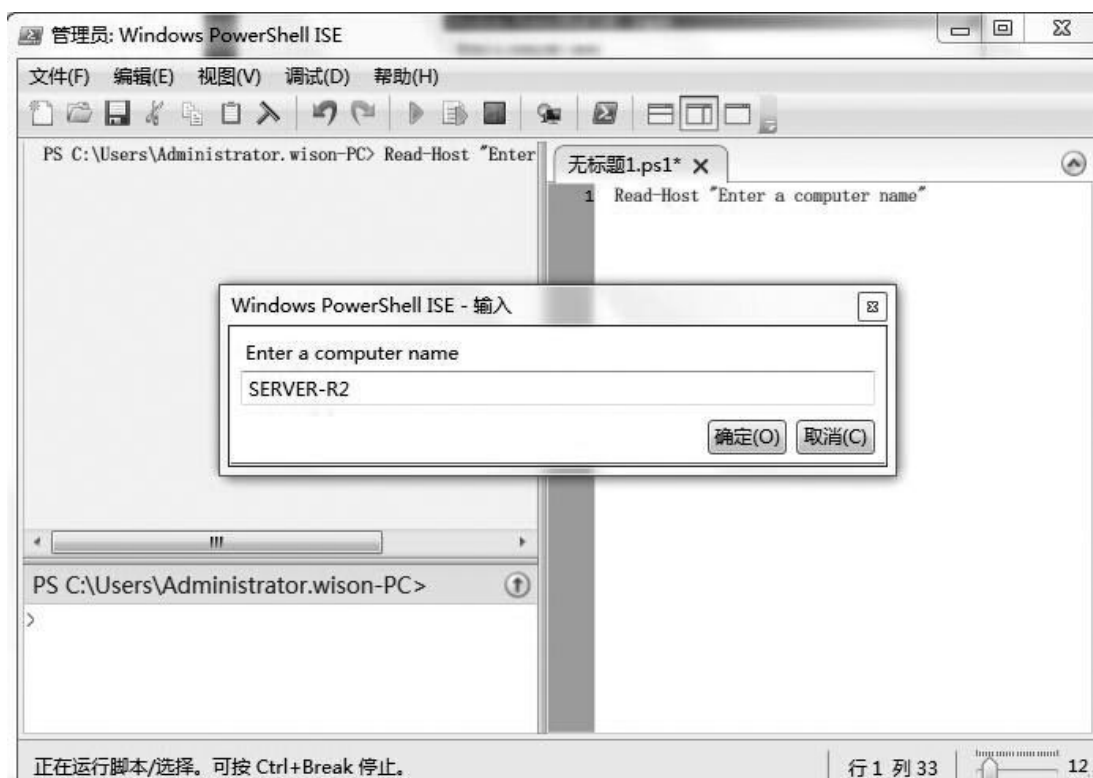


图19.2 第二版的ISE会为Read-Host命令弹出一个对话框

关于Read-Host命令也没什么好再多谈的了：它是一个很有用的Cmdlet，但是并不是一个让人很兴奋的Cmdlet。实际上，在大多数课堂讲解了Read-Host命令后，总有人会问我们：“是否有其他方法可以始终展现一个图形化的输入框？”很多管理员会给用户部署一些PowerShell脚本，但是又不希望用户必须在命令行界面输入信息（毕竟，这并不是很“Windows风格”）。我们想说的是可以实现，但是稍微复杂。最终的结果如图19.3所示。

为了创建一个图形界面的输入框，你必须借助于.Net Framework本身。使用下面的命令：

```
PS C:\> [void]
[System.Reflection.Assembly]::LoadWithPartialName('Microsoft.
➔VisualBasic')
```

你只需要在某个PowerShell会话执行一次即可，但是即使执行多次，也不会有什么影响。

该命令会载入.Net Framework中的一个组件Microsoft.VisualBasic，实际上PowerShell并不会自动载入该组件。该Framework组件包含了大量的VisualBasic核心的框架元素，其中就包括图形化输入框。

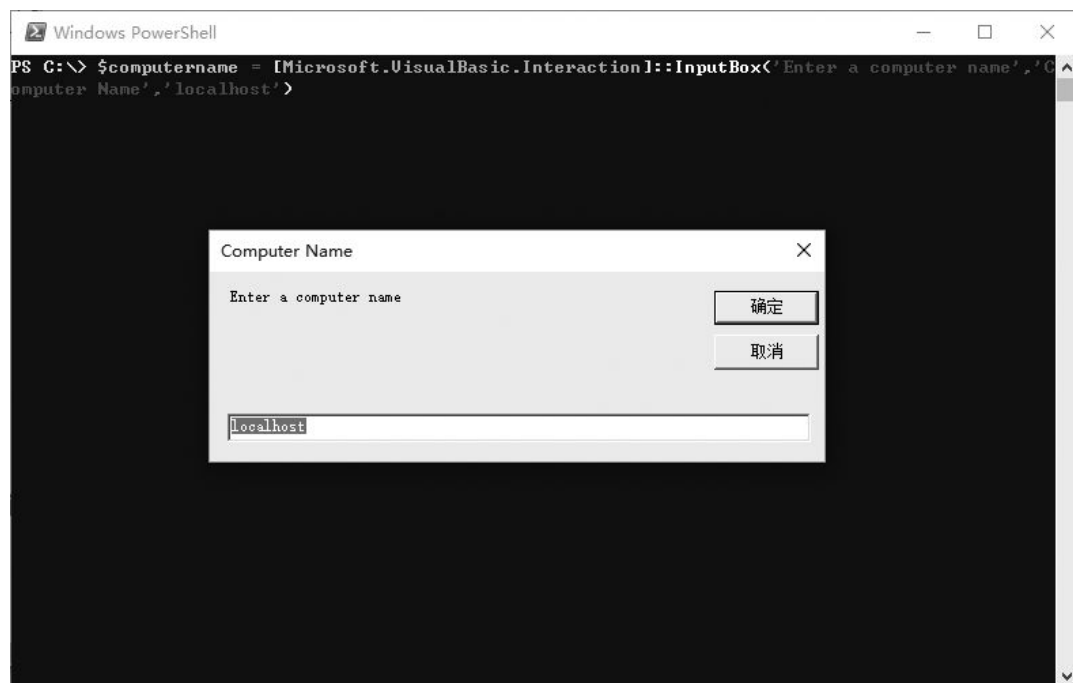


图19.3 在Windows PowerShell中创建一个图形化输入框

让我们看看该命令是怎么实现的：

- [Void]部分将命令的返回结果转化为[Void]类型。在前面的章节中，你已经学过如何转化整型数据；Void数据类型是一种特定的类型，意味着“抛弃产生的结果”。我们不想看到该命令的执行结果，所以我们将该结果转化为Void类型。实现该目的的另外一种方法是将该结果集通过管道传递给Out-Null。

- 接下来我们会访问**System.Reflection.Assembly**类型，该类型代表了我们的应用程序（在这里就是**PowerShell**）。我们将该类型名称放在一个方括号内，犹如我们申明了一个该类型的变量。但是我们这里并不是真正申明一个变量，而是用了两个冒号来访问该类型的静态方法。静态方法并不依赖于我们创建一个该类型的实例而存在。
- 我们这里使用的静态方法是**LoadWithPartialName()**，该方法会接收我们希望添加的**Framework**组件名称。

如果你觉得很难理解，也没关系；你可以照搬该命令，不需要理解它们的原理。一旦该**Framework**组件被载入，你可以通过下面的命令来使用它。

```
PS C:\> $ComputerName =  
[Microsoft.VisualBasic.Interaction]::InputBox('Enter a  
➔computer name','Computer Name','localhost')
```

在该示例中，我们再次使用了一个静态方法，这一次是**Microsoft.VisualBasic.Inter Action**类型。我们使用前面的命令将其载入到内存中。再次说明，如果你对“静态方法”感到很难理解，也没关系——直接照搬命令即可。

这里你可以修改的地方是**InputBox()**方法的三个参数。

- 第一个参数是提示框中的文本信息。
- 第二个参数是提示对话框的标题。
- 第三个参数——可以是空白或者完全省略，是你想显示在输入框中的默认值。

使用**Read-Host**命令可能比前面示例的步骤稍微简单，但是如果你仍然坚持使用对话框，该示例就说明了如何创建该对话框。

19.3 Write-Host命令

既然你可以收集输入信息，那么也会希望了解一些展示返回结果的方法。**Write-Host**命令就是其中的一种方法。这并不总是最好的一种

方法，但是你可以使用它，并且重要的是，你需要了解它的工作原理。

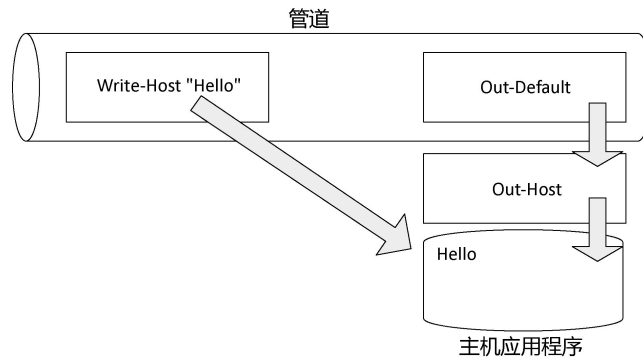


图19.4 Write-Host会绕开管道，直接写到主机应用程序的显示界面

如图19.4所示，Write-Host会和其他Cmdlet一样使用管道，但是它并不会放置任何数据到管道中。相反，它会直接写到主机应用程序的界面。正因为可以这样做，所以我们可以使用命令行中的-ForegroundColor和-BackgroundColor参数来将前景和背景设置为其他颜色。

```
PS C:\> Write-Host "COLORFUL!" -Fore Yellow -Back Magenta
COLORFUL!
```

动手实验： 你需要运行该命令来查看带有色彩的结果集。

注意： 不是每个使用PowerShell的应用程序都支持其他颜色，也并不是每个应用程序都支持整系列的颜色。当你尝试在某个应用程序中设置颜色时，通常会忽略掉不喜欢或者不能显示的颜色。这也是我们需要避免依赖于特定颜色的一个原因。

当需要展示一个特定的信息，比如使用其他颜色来吸引人们的注意力时，你应该使用Write-Host命令。但是针对使用脚本或者命令来产生常规的输出结果而言，这并不是一个恰当的方法。

例如，你永远都不应该使用Write-Host命令来手动格式化一个表格——你能找到更好的方法来产生输出结果，比如使用那些让PowerShell可以实现处理格式化功能的技巧。在本书中我们不会讲到

这些技巧，因为它们更多属于较为复杂的脚本以及工具制作领域。但是，你可以通过 *_Learn PowerShell Toolmaking in a Month of Lunches (Manning, 2012)* 来学习这些输出技巧的全部知识。针对产生错误信息、警告信息、调试信息等而言，**Write-Host**命令也不是最好的方法——再次申明，你可以找到更合适的方法来实现这些功能。当然本书中也会讲到这些。如果你恰当地使用PowerShell，那么你可能不会多次使用到**Write-Host**命令。

注意： 我们经常看到有人使用**Write-Host**命令来显示“温暖和模糊”的信息——比如“nowconnecting to Server-2”，“testing for folder”等。请不要这样做，有更恰当的方法来实现这些功能，就是**Write-Verbose**。

补充说明

我们将在第22章中深入讲解**Write-Verbose**以及其他的一些**Write Cmdlet**。但是如果你现在尝试使用**Write-Verbose**命令，你可能会很沮丧地发现该命令不会返回任何的结果，准确地说是默认情况下不会返回。

如果你计划使用**Write Cmdlet**，诀窍就是首先打开它们。例如，设置`$VerbosePreference= "Continue"`将会启用**Write-Verbose**，`$VerbosePreference="SilentlyContinue"`会截断其输出。你会看到针对**Write-Debug** (`$DebugPreference`) 和**Write-Warning** (`$WarningPreference`) 命令也存在类似的“Preference”变量。

在第22章中会介绍一种更酷的方法来使用**Write-Verbose**命令。

看起来使用**Write-Host**命令会更容易，如果你希望使用该命令，那么也可以。但是请记住，如果使用其他的**Cmdlet**，比如**Write-Verbose**命令，你会更加贴近PowerShell本身的使用方式，最终得到更一致的体验。

19.4 Write-Output命令

不像Write-Host命令，Write-Output命令可以将对象发送给管道。因为它不会直接写到显示界面，所以不允许你指定其他任何的颜色。实际上从技术来说，Write-Output（或者它的别名Write）根本不是设计出来展示结果的。正如我们所讲，它将这些对象发送给管道——也就是最终展示这些对象的管道。图19.5展现了对应的工作原理。

快速复习一下第10章中的知识点：如何将对象从管道传递给显示界面。下面就是最基本的过程。

- (1) Write-Output命令将String类型的对象Hello放入到管道中。
- (2) 因为管道中不存在其他对象，Hello会到达管道的最末端，也就是Out-Default命令的位置。
- (3) Out-Default命令将对象传递给Out-Host命令。

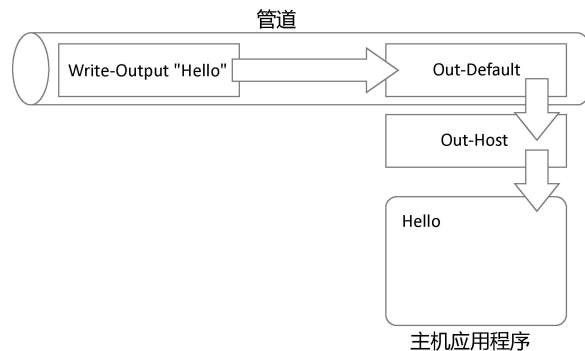


图19.5 Write-Output将对象放入管道，在某些情况下，最终会导致对象被展示出来

(4) Out-Host命令要求PowerShell的格式化系统格式化该对象。因为该示例中为简单的String对象，所以格式化系统会返回该String对象的文本信息。

- (5) Out-Host将格式化的结果集放在显示界面上。

执行的结果类似使用Write-Host命令的返回结果，但是该对象通过不同的路径到达最后阶段。该路径是非常重要的，因为在管道中可以包含其他的对象。例如，考虑下面的命令（欢迎你尝试执行该命令）：

```
PS C:\> Write-Output "Hello" | Where-Object { $_.Length -GT 10 }
```

你并没有看到该命令返回任何结果集，图19.6解释了其原因。“Hello”字符被放进管道。但是在它到达Out-Default命令之前，它必须经由Where-Object命令，该命令会去除长度（Length）属性小于或者等于10的对象。在该示例中，字符对象是“Hello”，所以此时该对象就会从管道中被筛选掉。由于在管道中不存在任何对象可以被传递给Out-Default，因此最终也就没有对象传递给Out-Host命令，那么也就不会显示任何的信息。

将前一个命令与下面的命令进行对比：

```
PS C:\> Write-Host "Hello" | Where-Object { $_.Length -GT 10 }  
Hello
```

这里所做的变更仅是使用Write-Host替换了Write-Output命令。这时，“Hello”字符会直接被传递给显示界面，而不会进入管道中。Where-Object命令并没有任何传入数据，因此也就不会有任何信息经由Out-Default和Out-Host展现出来。但是由于“Hello”字符已经被直接传递给显示界面，所以我们仍然可以看到它。

Write-Output命令看起来可能是新学习的命令，但是其实你一直都在使用它。它是PowerShell默认使用的一个Cmdlet。当你通知PowerShell去完成某项功能（但是又不是使用命令）时，PowerShell会在底层将你键入的任意信息传递给Write-Output命令。

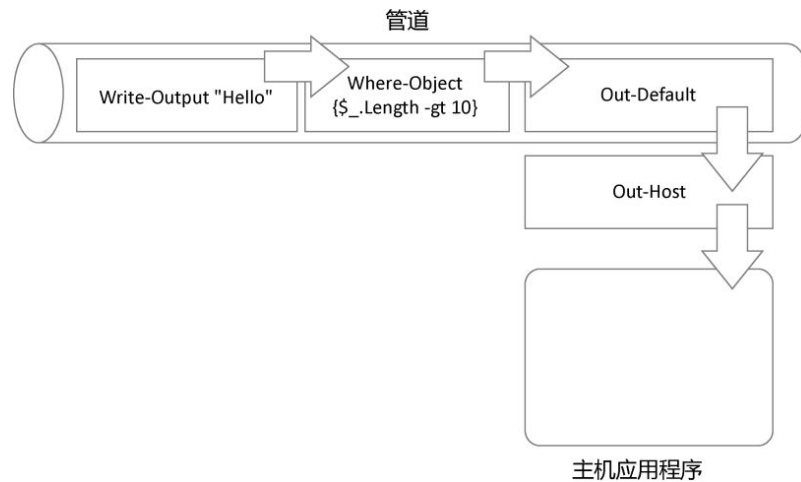


图19.6 将对象放进管道，也就意味着它们在显示之前可以被过滤掉

19.5 其他写入的方式

PowerShell中也存在其他方法来产生输出结果。这些方法都不会像Write-Host那样向管道写入某些信息，它们看起来更像是Write-Host命令。但是它们可以通过可被截断的方式产生结果。

PowerShell针对每种输出方法都有对应的内置配置变量。如果配置变量设置为“Continue”，那么我们即将展示给你的命令就会真正产生输出结果。如果配置变量被设置为“SilentlyContinue”，那么关联的输出命令就不会产生任何信息。表19.1包含了这些Cmdlet的列表。

表19.1 可选的输出Cmdlet

Cmdlet	作用	配置变量
Write-Warning	显示警告信息，默认会以黄色字体显示，同时前面带有“警告：”字样	\$WarningPreference（默认为Continue）
Write-Verbose	显示详细信息，默认会以黄色字体显示，同时前面带有“详细信息：”字样	\$VerbosePreference（默认为SilentlyContinue）
Write-Debug	显示调试信息，默认以黄色字体显示，同时前面带有“调试：”字样	\$DebugPreference（默认为SilentlyContinue）
Write-Error	产生一个错误信息	\$ErrorActionPreference（默认为Continue）

`Write-Error`命令会有点不一样，因为它会将错误信息写入PowerShell的错误流中。

另外，PowerShell还存在一个Cmdlet `Write-Progress`，该Cmdlet可以展示进度条，但是实现原理完全不一样。你可以阅读其帮助文档来获取更多的信息以及示例。本书中不会涉及该命令。

为了使用这些Cmdlet，首先你需要确保关联的配置变量设置为“Continue”。（如果上面列表中的两个Cmdlet的配置变量保留默认值`SilentlyContinue`，你不会看到任何的输出结果。）之后，就可以使用该Cmdlet来输出一些信息。

注意： 分PowerShell的主机应用程序会在不同的位置展现这些Cmdlet的输出信息。比如在PrimalScript中，调试信息会写入到另外一块输出窗格中，而不是脚本的主输出窗格，这样可以更容易将调试信息独立开来进行分析。在本书中，我们不会深入讲解调试相关的知识，但是如果你感兴趣，可以阅读PowerShell帮助文档中该Cmdlet对应的部分。

19.6 动手实验

注意： 对于本次动手实验环节，需要运行3.0版本或者之后版本的PowerShell。

`Write-Host`和`Write-Output`命令可能使用起来更为棘手。试试看，你可以完成下面列表中的几个任务。如果无法完成其中某些任务，那么可以参考MoreLunches.com网站上的示例答案。

1. 使用`Write-Output`命令来返回100除以10的结果。
2. 使用`Write-Host`命令来返回100除以10的结果。
3. 提示用户输入姓名，然后以黄色字体显示该姓名。
4. 提示用户输入姓名，并且仅当长度大于5时才显示该姓名。请使用单行命令完成——不要使用变量。

这就是本章动手实验环节的全部任务。因为这些Cmdlet都很简单，我们希望你能自行花更多的时间来测试它们。请保证一定要测试——在接下来的部分，我们会提供一些建议。

动手实验：完成本章节的动手实验环节后，请尝试完成本书附录中的实验回顾3。

19.7 进一步学习

请花费一定的时间来熟悉本章中所有的Cmdlet。确保你可以通过这些Cmdlet显示详细信息，接收输入数据，甚至可以显示图形的输入框。从现在起，你将会使用本章中所讲的Cmdlet，因此你应该阅读它们对应的帮助文档，甚至简单记下它们的简单语法提示，以便后续查找。

第20章 轻松实现远程控制

在第13章中，我们介绍了PowerShell的远程控制功能。在第13章中，你使用了实现远程控制的两个主要Cmdlet——**Invoke-Command**和**Enter-PSSession**——用于分别实现一对一以及一对多的远程控制。这两个命令的工作原理是创建一个远程连接，完成你指定的工作，然后关闭连接。

上面的方式并无不妥，但每次不断指定计算机名称、凭据、备用端口号等是一件非常麻烦的事情。在本章中，我们将查看更加简单、更可重用的方式实现远程控制。你还可以学到迟早会用得到的使用远程控制的第三种方式。

20.1 PowerShell远程控制稍微容易一点

每次使用**Invoke-Command**或**Enter-PSSession**命令连接远程计算机时，你至少需要指定计算机名称（或多个名称，如果你需要在多台计算机上调用命令）。根据具体环境的不同，你可能还需要指定备用凭据，这意味着需要你输入密码。你或许还需要指定备用端口或身份验证机制，这取决于你的组织是如何配置远程控制的。

上面的选项并没有很难指定的，但不断重复输入却让人感到乏味。幸运的是，我们知道一种更好的方法：可重用会话。

20.2 创建并使用可重用会话

会话是一个在你的PowerShell副本与远程PowerShell副本之间的持久化连接。当一个会话处于活动状态时，你的计算机与远程计算机都会划分出一小部分用于维护连接的内存和处理器时间，还有非常少一部分与连接相关的网络流量。PowerShell维护一个所有已打开的会话列表，你可以使用这些会话调用命令或进入远程Shell。

你可以通过**New-PSSession**这个Cmdlet创建一个新的会话，指定一个或多个计算机名称。如果需要，还可以指定备用用户名称、端口以

及身份验证机制等。结果是一个存在PowerShell内存中的会话对象：

```
PS C:\> new-pssession -computername server-r2,server17,dc5
```

通过Get-PSSession获取创建好的会话：

```
PS C:\> get-pssession
```

虽然上面的方法可以奏效，但我们更倾向于创建Session后立刻将其存入变量。例如，Don有三个基于IIS的Web服务器，它需要定期通过Invoke-Command命令配置这些服务器。为了让过程变得简单，它将这些会话存入特定变量：

```
PS C:\> $iis_servers = new-pssession -comp web1,web2,web3  
➡ -credential WebAdmin
```

请永远不要忘记这些会话会消耗资源。如果关闭Shell，那么这些会话也会随之关闭，但如果你不是频繁使用这些会话，那么即使你希望使用同一个Shell完成其他任务，手动关闭这些会话也是不错的主意。

使用Remove-PSSession这个Cmdlet关闭会话。比如说，只关闭连接到IIS的会话，可以使用下面的命令：

```
PS C:\> $iis_servers | remove-pssession
```

或者，如果希望关闭所有处于开启状态的会话，使用下面的命令：

```
PS C:\> get-pssession | remove-pssession
```

就是这么简单。

一旦成功建立会话后，你该如何使用这些会话？在接下来几小节中，我们假设你已经创建了一个名称为`$sessions`的变量，并至少包含两个会话。我们使用`localhost`和`Server-R2`（你应该指定为符合你具体环境的计算机名称）。使用`Localhost`并不是一个语法糖：PowerShell会开启一个真正指向本机PowerShell副本的远程会话。请记住，只有在所有连接到的计算机上都启用了远程控制时，远程连接才会生效。如果还未启用远程控制，请返回第13章。

动手实验： 跟随上面的步骤并运行这些命令，确保使用有效的计算机名称。如果你只有一台计算机，请使用计算机名称和`localhost`。

补充说明

有一个允许你使用一个命令创建多个会话，并将每个会话赋值给唯一变量的语法（而不是像之前的示例，将其全部塞入一个变量）：

```
$s_server1,$s_server2 = new-pssession -computer server-r2,dc01
```

该语法将连接到`Server-R2`服务器的会话存入变量`$s_server1`，将连接到`DC01`服务器的会话存入`$s_server2`，这使得独立使用不同的会话变得简单。

但是请小心使用：我们曾见过会话的顺序和计算机名称的顺序不完全一致，导致`$s_server1`最终包含连接到`DC01`而不是`Server-R2`的会话。你可以将变量内容显示出来，从而查看会话连接到哪一台计算机。

下面的代码用于建立好会话并使其运行：

```
PS C:\> $sessions = New-PSSession -comp SERVER-R2,localhost
```

请记住，我们已经在这些计算机上启用了远程控制，并且这些计算机处于同一个域。如果你希望回忆起如何启用远程控制，请再次查看第13章。

20.3 利用Enter-PSSession命令使用会话

希望你回忆起第13章，Enter-PSSession命令是用于进入远程计算机一对一的交互式Shell所用的命令。该命令的参数可以是一个会话对象，而不是具体的计算机名称。由于\$session变量中包含两个会话对象，我们必须通过索引指定使用其中哪一个会话对象（你在第18章中学到过）：

```
PS C:\> enter-pssession -session $sessions[0]
[server-r2]: PS C:\Users\Administrator\Documents>
```

可以看到命令提示符已经改变，表示我们已经在控制远程计算机。Exit-PSSession命令用于帮助我们返回到本地提示符，但远程命令并不会中断，以便于后续使用。

```
[server-r2]: PS C:\Users\Administrator\Documents>exit-pssession
psc:\>
```

或许你很难记起具体哪一个索引号对应哪一个计算机名称。如果是这种情况，你可以利用会话对象的属性进行区分。例如，当我们将\$sessions对象通过管道传递给Gm命令时，我们可以得到如下输出结果：

```
PS C:\> $sessions | gm

      TypeName: System.Management.Automation.Runspaces.PSSession

Name      MemberType      Definition
-----
Equals    Method          bool Equals(System.Object
obj)
GetHashCode Method          int GetHashCode()
```

GetType	Method	type GetType()
ToString	Method	string ToString()
ApplicationPrivateData	Property	
System.Management.Automation.PSPr...		
Availability	Property	
System.Management.Automation.Runs...		
ComputerName	Property	System.String ComputerName
{get;}		
ConfigurationName	Property	System.String
ConfigurationName {...		
Id	Property	System.Int32 Id {get;}
InstanceId	Property	System.Guid InstanceId
{get;}		
Name	Property	System.String Name {get;set;}
Runspace	Property	
System.Management.Automation.Runs...		
State	ScriptProperty	System.Object State
{get=\$this.Ru...		

在上面的输出结果中，你可以看到会话对象包含一个名为 **ComputerName** 的属性。这意味着你可以筛选出该会话：

```
PS C:\> enter-ssession -session ($sessions |
➔where { $_.computername -eq 'server-r2' })
[server-r2]: PS C:\Users\Administrator\Documents>
```

这个语法的处境比较尴尬，因为如果你需要使用变量中的一个会话，但记不住其中的会话索引号，或许你会更容易忘记使用变量。

即使你将会话对象存于变量中，这些会话依然被存于 **PowerShell** 的一个打开会话的主列表中。这意味着你可以通过 **Get-PSSession** 访问这些会话：

```
PS C:\> enter-ssession -session (get-ssession -computer server-
r2)
```

Get-PSSession 将会获取名称为 **Server-R2** 的计算机，并将其传递给 **Enter-PSSession** 命令的 **-Session** 参数。

当我们第一次发现这个技巧时，我们非常震惊，但这也让我们更进一步。我们找出**Enter-PSSession**的完整帮助并仔细阅读**-Session**参数。下面是我们所看到的：

```
-Session <PSSession>
    Specifies a Windows PowerShell session (PSSession) to use for
the
    interactive session. This parameter takes a session object.
You can
    also use the Name, InstanceID, or ID parameters to specify a
    PSSession.

    Enter a variable that contains a session object or a command
that
    creates or gets a session object, such as a New-PSSession or
Get-
    PSSession command. You can also pipe a session object to
Enter-
    PSSession. You can submit only one PSSession with this
parameter. If
    you enter a variable that contains more than one PSSession,
the
    command fails.
    When you use Exit-PSSession or the EXIT keyword, the
interactive
    session ends, but the PSSession that you created remains open
and a

    available for use.
    Required?                false
    Position?                1
    Default value
    Accept pipeline input?    true (ByValue, ByPropertyName)
Accept wildcard characters? True
```

如果你回想一下第9章的内容，你将会发现在帮助末尾的管道输入信息非常有趣。该信息告诉我们，**-Session**参数可以从管道接受一个**PSSession**对象。我们知道**Get-PSSession**命令会生成**PSSession**对象，所以下述语法也可以生效。

```
PS C:\> Get-PSSession -ComputerName SERVER-R2 | Enter-PSSession
```

```
[server-r2]: PS C:\Users\Administrator\Documents>
```

该命令的确可以生效。我们认为，就算你已经将所有会话存入一个对象中，使用该方式是一种更加优雅的获取单个对象的方式。

提示 为了方便，将会话存入一个变量是可以的。但请记住，PowerShell已经保存了所有已打开会话的列表；将这些会话存入变量，只有在你需要一次性引用多个会话时才有用，正如你将在下一小节所见。

20.4 利用Invoke-Command命令使用会话

Invoke-Command命令展示了Session对象的价值，你习惯于用该命令将一个命令（或一个完整的脚本）并行在多个远程计算机上执行。我们已经将所有的会话存储在\$Session变量中，我们可以通过下面的命令轻松将多个计算机作为目标。

```
PS C:\> invoke-command -command { get-wmiobject -class  
win32_process }  
➡ -session $sessions
```

注意，我们将一个Get-WmiObject命令发送到远程计算机。我们本可以选择使用Get-WmiObject命令自带的-computername参数，但是由于下面4个原因，我们没有这么做。

- 远程控制通过一个预定义的端口进行传输，WMI却不是。远程控制因此针对在防火墙后的计算机更加容易使用，这是由于更容易开启必要的防火墙例外。微软Windows防火墙为包含必要的状态检测使得WMI随机端口选择（也就是端点匹配）可以正常工作的WMI提供了一个特定的例外，但对于其他第三方防火墙产品来说却难以管理。通过远程控制就容易很多，因为只有一个端口。
- 将所有的进程传输到本地费时费力。使用Invoke-Command这个Cmdlet，可以让每一个计算机完成各自的工作，并将结果返回。
- 远程控制并行执行，默认可以连接最多32台计算机。WMI顺序执行，一次只能在一台计算机上执行。
- 我们无法通过Get-WmiObject使用我们预定义的会话对象，但可以通过Invoke-Command使用。

注意：在PowerShell v3中，新的CIM Cmdlet（比如说Get-CimInstance）并不像Get-WmiObject那样有一个-computerName参数。新的Cmdlet被设计的本意就是，如果希望在远程计算机上执行，请通过Invoke-Command将其发送过去。

Invoke-Command的-Session参数也可以通过括号命令提供，正如我们在之前章节对计算机名称所做的那样。举例来说，下面的语句会将命令发送给计算机名称以“loc”开头的已连接会话：

```
PS C:\> invoke-command -command { get-wmiobject -class  
win32_process }  
➡ -session (get-pssession -comp loc*)
```

你或许会期望Invoke-Command可以从管道中接收会话对象，就像Enter-PSSession命令那样。但通过查看Invoke-Command的完整帮助，会发现它并不支持这种使用管道的技巧。倒霉，但之前使用括号表达式的示例提供了同样的功能，而无需太复杂的语法。

20.5 隐式远程控制：导入一个会话

隐式远程控制是对我们来说最酷、最有用的功能之一——可能是在任何操作系统的命令行界面中迄今为止最酷、最有用的功能。但不幸的是，该功能并未记入PowerShell文档。当然，那些必要的命令都有良好的文档，但这些必要命令共同汇集在一起形成的这个强大功能却没有在文档中被提及。所幸，我们在本文中对该功能进行了阐述。

让我们重新回顾一下场景：你已经知道微软已经针对Windows和其他产品发行越来越多的模块和插件，但由于各种各样的原因，你无法将这些模块安装在本地计算机上。在Windows Server 2008 R2上第一次发行的活动目录（ActiveDirectory）模块就是一个很好的示例：该模块只存在于Windows Server 2008 R2以及安装远程服务器管理工具（Remote Server Administration Tools, RSAT）的Windows 7上。如果计算机的操作系统是Windows XP或Windows Vista呢？是否就无法安装了？当然不是，你可以使用隐式远程控制。

让我们通过一个示例来查看完整的过程。

```

PS C:\> $session = new-pssession -comp server-r2
PS C:\> invoke-command -command
➡ { import-module activedirectory }
➡ -session $session
PS C:\> import-pssession -session $session
➡ -module activedirectory
➡ -prefix rem

ModuleType Name
-----
Script tmp_2b9451dc-b973-495d... {Set-ADOrganizationalUnit, Get-ADD...

```

下面是本示例的解释。

① 首先，通过与一台装有活动目录模块的远程计算机建立一个会话。我们需要该计算机装有**PowerShell v2**或更新版本（在**Windows Server 2008 R2**以及更新版本的操作系统上），我们必须启用该计算机的远程控制。

② 我们告诉远程计算机导入其本地的活动目录模块。这只是一个示例。我们当然可以选择载入任意模块，甚至是在需要时添加一个**PSSnapin**。由于会话处于打开状态，该模块将一直在远程计算机上处于被载入状态。

③ 我们接下来告诉我们的计算机从远程会话中导入命令。我们只需要在活动目录模块中的命令，并在每个命令的名词部分加入“**rem**”前缀。这使得我们可以更容易跟踪远程命令。这还意味着从远程会话导入的命令不会与已经在本地**Shell**中导入的命令冲突。

④ **PowerShell**在本地计算机创建一个临时模块，用于代表远程命令。这些命令并不是被复制过来的；**PowerShell**为其创建了指向远程计算机的快捷方式。

现在我们就可以运行活动目录模块的命令了，甚至是使用帮助命令。我们使用**New-remADUser**来代替**New-ADUser**，这是由于我们在命令的名词部分添加了前缀“**rem**”。该命令在我们关闭**Shell**或关闭与远程连接的会话之前一直存在。当我们打开一个新的**Shell**时，我们必须重复上述过程来重新活动访问远程命令的权限。

当我们运行这些命令时，它们并不是在我们本地计算机上执行，而是隐式地在远程计算机上执行。在远程计算机上执行完成后，将结

果发送给本地计算机。

我们可以想象出这样一个世界：我们永远不需要在本地计算机安装管理工具，这将避免多少麻烦。今天，你需要在本地操作系统上安装可运行的工具，并与你尝试管理的远程计算机进行通信——这使得匹配所有远程与本地的功能几乎不可能。而在未来，你无须再这么做。你将只需要使用隐式远程控制。服务器将通过PowerShell将其管理功能作为一个服务开放出来。

接下来到了坏消息时间：通过隐式远程连接获取到本地计算机的结果是反序列化的结果，这意味着对象的属性将会复制到一个XML文件中，以便通过网络进行传输。用这种方式收到的对象不会包含任何方式。在大多数情况下，这并不是一个问题。但你希望以编程的方式使用模块或插件时，这些模块或插件对隐式远程控制的支持就不会那么好了。我们希望该限制不会影响到你，这是由于对方法的依赖违反了一些PowerShell的设计实践。如果你用到了这些对象，则无法通过隐式远程控制的方式使用它们。

20.6 断开会话

PowerShell v3对远程控制引入了两项提升。

首先，会话不再那么脆弱，意思是在网络闪断或其他传输中断的情况下，会话不会断开。即使在没有显式使用会话对象时，你也可以用到这项提升。即使你在使用类似Enter-PSSession和它的-ComputerName参数时，从技术角度，你也是在底层使用了会话。因此，你获得了更稳定的连接。

在第三版中，另一项功能是你必须显式使用的：断开会话。比如你正在以用户Admin1（是Domain Admins组成员）的身份连接到名称为Computer1的计算机上，并创建一个连接到名称为Computer2的连接：

```
PS C:\> New-PSSession -ComputerName COMPUTER2
```

Id	Name	ComputerName	State
--	-----	-----	-----

然后你就可以关闭连接。该操作仍然是在**Computer1**上进行的。当你完成该操作后，它会将两台计算机之间的连接断开，但会在**Computer2**上保留一份PowerShell的副本。注意，你可以通过指定Session的ID号完成该操作，该ID号会在你第一次创建Session时显示：

```
PS C:\> Disconnect-PSSession -Id 4
```

Id	Name	ComputerName	State
4	Session4	COMPUTER2	Disconnected

上面的内容值得你深入考虑——你在**Computer2**上保留一份PowerShell的副本处于运行状态。因此为其分配一个适用的超时时间就变得很重要。在PowerShell早期的版本中，断开连接的Session将会被丢弃，所以无需清理工作。在第三版中，未被回收的会话可能会导致一些问题，这意味着你必须负责起回收工作。

但最酷的地方在于，我们可以登录到另一台计算机，也就是**Computer3**上，用同样的域账号Admin1，并获取运行在**Computer2**上的会话列表。

```
PS C:\> Get-PSSession -computerName COMPUTER2
```

Id	Name	ComputerName	State
4	Session4	COMPUTER2	Disconnected

非常简单明了，不是吗？如果你以其他用户的身份登录，就无法看到这些会话。即使该身份为管理员，你也只能看到在**Computer2**上创建的会话。既然已经看到了，那么你就可以重新连接。

```
PS C:\> Get-PSSession -computerName COMPUTER2 | Connect-PSSession
```

Id	Name	ComputerName	State
----	------	--------------	-------

我们花一些时间讨论管理这些会话。在PowerShell的WSMAN:Drive, 你可以发现大量可以帮助你管控已断开会话的设置。你还可以通过组策略对大多数配置进行中心化管理。需要寻找的关键设置如下。

在WSMan:\localhost\Shell下:

- **-IdleTimeout**指定当远程Shell中没有用户活动时, 远程 Shell 将保持打开状态的最长时间。在指定的时间过后, 远程 Shell 将被自动删除。默认值是2 000小时, 活84天。
- **-MaxConcurrentUsers**指定可以在同一计算机上通过远程 Shell 同时执行远程操作的最大用户数。
- **-MaxShellRunTime**指定会话可以打开的最长时间。默认值为无限。请记住, IdleTimeout参数可以覆盖该参数。
- **-MaxShellsPerUser**指定任何用户可以在同一系统上远程打开的并发 Shell 的最大数目。将该值与MaxConcurrentUsers 相乘, 可以得到计算机上所有用户最大会话数量的值。

在WSMan:\localhost\Service下:

- **-MaxConnections** 设置连接到整个远程控制架构下的连接数上限。即使你设置了每个用户可运行的Shell数量或上限值的用户, MaxConnections也会限制传入连接。

作为一个管理员, 你明显比普通用户需要更高的责任心。你需要负责跟踪会话, 尤其是你需要断开连接和重新连接。设置合理的超时时间, 可以确保Shell的会话不会长时间闲置。

20.7 动手实验

注意: 对于本次动手实验来说, 你需要运行PowerShell v3或更新版本PowerShell的计算机。如果你只有一个客户端版本的计算机 (运行Windows 7或Windows 8), 你就无法完成本实验中的第6至9步。

为了完成本次动手实验，你需要两台计算机：一台作为远程控制的控制端，另一台作为远程控制的接收端。如果你只有一台计算机，使用计算机名称对其进行远程控制。这种方式的体验和真正的远程连接非常类似。

提示 在第1章中，我们提到了一个在CloudShare.com中的多计算机虚拟环境。你可以找到其他类似的基于云计算的虚拟主机。通过使用CloudShare，我们无须部署Windows操作系统，这是由于该服务已经提供了供我们使用的模板。你当然需要为此服务付费，且该服务并不是对所有的国家可用。但如果你可以使用该服务，在本地没有环境时，这是获得一个实验环境的极佳方式。

1. 在Shell中关闭所有已打开的连接。
2. 建立一个连接到远程计算机的会话，并将会话存入一个命名为`$session`的变量。
3. 利用`$session`变量建立一个一对一到远程计算机的远程控制Shell会话。
4. 将`Invoke-Command`命令与`$session`变量结合使用获取远程计算机上的服务列表。
5. 利用`Invoke-command`与`Get-PSSession`命令从远程计算机上获取最近20条远程安全事件日志条目。
6. 利用`Invoke-Command`与`$session`变量在远程计算机上载入`ServerManager`模块。
7. 将`ServerManager`模块的命令由远程计算机导入到本地计算机，并使得“rem”成为命令名词部分的前缀。
8. 运行刚刚导入的`Get-WindowsFeature`命令。
9. 关闭储存在`$session`变量中的会话。

注意： 多亏了PowerShell v3中的新功能，你还可以利用`Import-Module`命令一步完成步骤6和步骤7。请随意查看该命令的帮助文档，看看你是否能想出如何从远程计算机导入一个模块。

20.8 进一步学习

快速盘点一下你的环境： 包含哪些启用PowerShell的产品？

Exchange Server? SharePoint Server? VMware vSphere? System Center Virtual Machine Manager? 上述产品或其他产品都包括PowerShell模块或插件，其中大多数插件或模块都可以通过PowerShell远程控制进行访问。

第21章 你把这叫作脚本

目前为止，你已经可以通过PowerShell的命令行界面完成本书中的所有内容。但你仍然没有写过一行脚本。这对我们来说是很大的问题。这是因为我们见过很多管理员害怕写脚本，认为写脚本是一种编程方式并觉得学习写脚本得不偿失。所幸，你已经看到在不成为程序员的前提下使用PowerShell所能完成的工作。

但在此刻，你可能还会感觉不断重复输入同样的命令是一件非常枯燥的事情。你是对的，所以在本章我们将会深入PowerShell脚本——当然，你仍然无须成为程序员。脚本的作用仅仅是为了减少不必要的重复输入。

21.1 非编程，而更像是批处理文件

大多数Windows管理员曾经或是时不时地创建一个命令行批处理文件（通常以.BAT或.CMD作为文件扩展名）。该文件本质上不过是一个简单的、可以用Windows记事本编辑的文本文件，该文件包含按照指定顺序排列的可执行命令列表。从技术上讲，你把这些命令叫作脚本，就像好莱坞电影的剧本那样用于告诉演员（你的计算机）该如何按照顺序说台词和表演。但批处理文件看上去并不像是编程语言，这部分是由于cmd.exe Shell语言本身过于简单，难以编写非常复杂的脚本。

PowerShell脚本——如果你愿意或者也可以称之为批处理文件——以类似的原理工作。仅仅是将你希望运行的命令列出来，Shell将会以指定的顺序执行这些命令。你可以通过将命令从宿主窗口中复制到文本文件中来创建一个脚本。当然，记事本是一个非常不好用的文本编辑器。我们希望更倾向使用PowerShell ISE，或者诸如PowerGUI、PrimalScript或PowerShell Plus之类的第三方编辑器。

ISE实际上使用起来和使用交互性Shell并无不同。当使用ISE的脚本编辑器窗口时，只需输入命令或希望运行的命令，并单击在工具栏中的“运行”按钮执行这些命令。单击“保存”按钮，你将可以在不复制粘贴任何命令的情况下创建一个脚本。

21.2 使得命令可重复执行

PowerShell脚本背后的理念，首先是使得重复执行特定命令变得简单，而无须每次手动重复输入命令。既然如此，我们需要想出一个你能够一遍遍重复执行的命令，并使用该示例贯穿本章。我们希望该示例有合适的复杂度，所以我们以WMI开始并添加一些筛选条件、排序规则以及其他内容。

此时，我们需要转换使用PowerShell ISE而不是标准的控制台窗口。这是由于通过ISE将我们的命令转为一个脚本变得更加容易。坦白讲，ISE使得输入复杂命令变得更加容易。这是因为可以使用全屏的编辑器而不是在控制台宿主上输入单行命令。

下面是我们的命令。

```
Get-WmiObject -class Win32_LogicalDisk -computername localhost  
➔ -filter "drivetype=3" |  
Sort-Object -property DeviceID |  
Format-Table -property DeviceID,  
@{label='FreeSpace(MB)';expression={$_.FreeSpace / 1MB -as [int]}},  
@{label='Size(GB)';expression={$_.Size / 1GB -as [int]}},  
@{label='%Free';expression={$_.FreeSpace / $_.Size * 100 -as [int]}}
```

提示： 请记住，你可以使用name而不是label，这两个属性都可以简写为n或l。但L的小写形式看上去非常像数字1，所以请小心。

图21.1展示了我们如何在ISE中输入该命令。注意，我们通过在工具栏按钮距离左边很远的“在顶部显示脚本窗格”按钮选择了双窗格布局。另外注意，我们将命令格式化为每一个物理行以逗号或管道操作符结尾。这么做可以让Shell识别这个多行脚本是一个单个、单行的命令。你也可以在控制台宿主中这么做，但这种格式由于具有更好的可读性，因此在ISE中尤其有效。另外注意，我们使用的是完整Cmdlet名称和参数名称并显式指定了参数名称，而不是使用位置参数。上面我们所做的一切都是为了使脚本具有更好的可读性，以便其他人很快可以接手。此外，当我们未来忘了当初脚本的意图时，可以很快想起来。

我们通过单击在工具栏的绿色运行按钮运行命令（也可以按快捷键F5），对命令进行测试，输出结果显示命令正常工作。下面是在ISE中一个巧妙的技巧：你可以选中命令的一部分并按F8键，从而只运行选中部

分的命令。由于我们已经格式化了命令，因此每一个物理行只有一个单独命令，这使得分步测试命令变得更加容易。我们可以选中并单独运行第一行命令。如果输出结果符合预期，我们可以选中第一行和第二行命令并运行。如果这部分也能正常工作，那么我们就可以运行整个命令。

此时，我们就可以保存命令——现在就可以把保存后的命令称为脚本。我们可以将其另存为`Get-DiskInventory.ps1`。我们以“动词-名词”这样的Cmdlet风格名称命名该脚本。你可以看到该脚本是如何开始像Cmdlet一样工作的，这也是使用Cmdlet风格名称的原因。

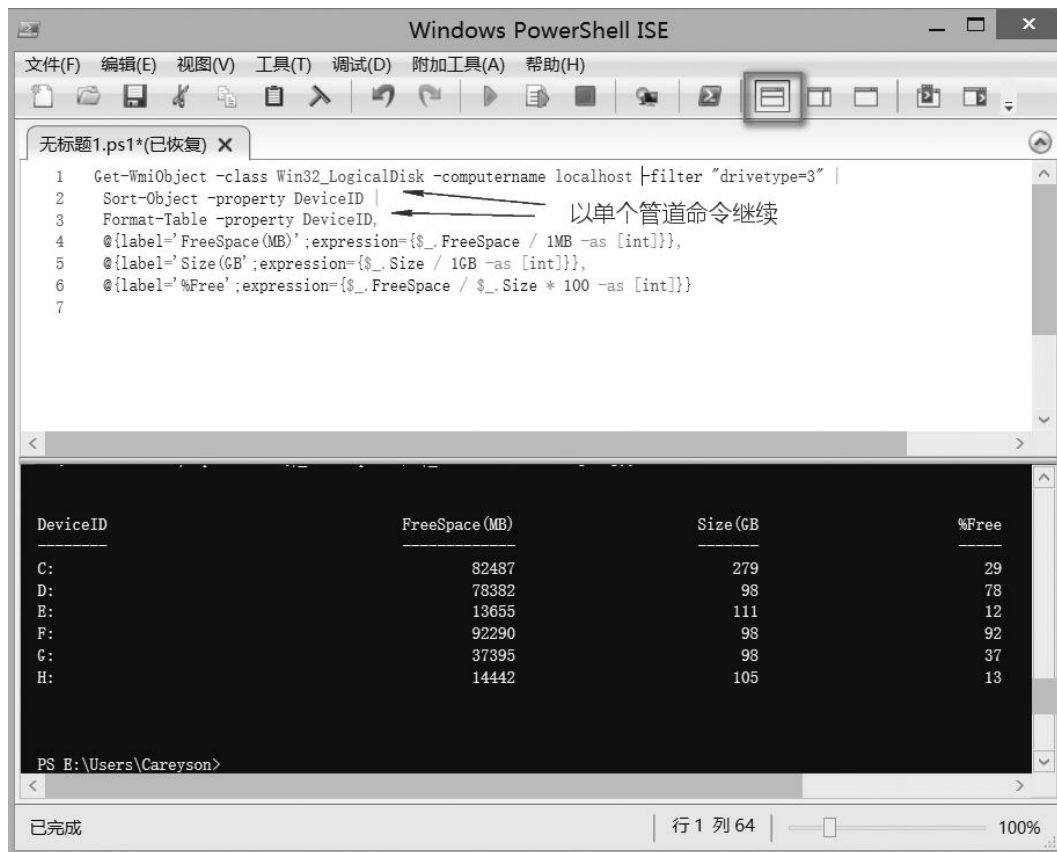


图21.1 在ISE中输入并运行一个命令

动手实验： 我们假设你已经完成了第14章并设置了更加自由的执行策略。如果你还未这么做，那么请返回第17章完成其动手实验部分，这样该脚本就可以在你的PowerShell副本下运行。

21.3 参数化命令

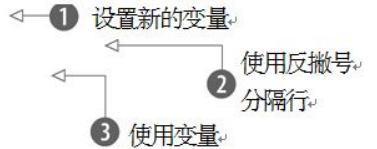
当你考虑到一遍遍运行同一个命令时，你或许会意识到命令的某些部分在每次运行时都可能产生变化。例如，假设你将**Get-DiskInventory.ps1**脚本给了一个缺乏PowerShell使用经验的同事。该脚本是一个比较复杂且难以输入的命令，你的同事非常感激你将其封装为一个易于运行的脚本。但是，作为该脚本作者，你发现该脚本只能够在本地计算机上运行。你当然可以想象得出，你的一些同事或许希望从一台或多台远程计算机上获取磁盘信息。

一个可能的解决方案是让他们打开脚本，并修改**-computer-name**参数值。但这个操作可能对它们来说有点难度，且修改脚本可能导致改错地方从而破坏脚本。因此为他们提供一个标准方法，使得他们可以传入不同的计算机名称（或名称集合）将是一种更好的方式。在此阶段，你需要识别出命令执行时可能需要变更的部分，并用变量替换这部分。

既然我们仍然处于测试脚本阶段，我们暂时将计算机名称变量设置为静态值。下面是修改后的脚本。

代码清单21.1 Get-DiskInventory.ps1，包含一个参数的命令

```
$computername = 'localhost'
Get-WmiObject -class Win32_LogicalDisk `
  -computername $computername `
  -filter "drivetype=3" |
Sort-Object -property DeviceID |
Format-Table -property DeviceID,
  @{label='FreeSpace(MB)';expression={$_.FreeSpace / 1MB -as [int]}},
  @{label='Size(GB)';expression={$_.Size / 1GB -as [int]}},
  @{label='%Free';expression={$_.FreeSpace / $_.Size * 100 -as [int]}}
```



我们在此完成了三件事，其中两件关于功能，另一件是格式美化。

- 我们添加了一个变量**\$computername**，将其值设置为**localhost** ❶。我们注意到，大多数PowerShell命令使用名称为**-computerName**的参数接受计算机名称。我们希望保留这种传统，这也是为什么我们将变量命名为**\$computername**。
- 我们将**-computerName**参数值替换为我们定义的变量 ❷。当前，该脚本和之前的脚本功能完全一样（并且经过测试的确一样），这是由于我们已经将**localhost**值赋予**\$computerName**变量。

- 我们在`-computerName`参数和其值后面添加了反撇号```。这是转义符号，该符号用于告诉PowerShell下一个物理行是之前命令的一部分。当行以管道操作符或逗号结尾时无须使用转义符号，但需要按照本书的代码结构组织代码。这里我们需要在管道操作符之前分隔行，因此只能在行末尾使用反撇号。

我们再次仔细检查并运行脚本，从而确保脚本仍然可以正确工作。在每次对脚本进行任何变更时，我们总是会这么做，以便确保没有引入新的误输入或其他错误。

21.4 创建一个带参数的脚本

既然我们已经识别出了脚本中每次执行可能变化的部分，那么我们就需要提供一种让其他人赋予这些元素新值的方式。换句话说，我们需要将被赋予常量的`$computername`变量转变为一个输入参数。

PowerShell中创建一个带参数的脚本非常简单。

代码清单21.2 Get-DiskInventory.ps1，包含一个输入参数

```
param (
    $computername = 'localhost'
)
Get-WmiObject -class Win32_LogicalDisk -computername $computername `
    -filter "drivetype=3" |
    Sort-Object -property DeviceID |
    Format-Table -property DeviceID,
        @{label='FreeSpace(MB)';expression={$_.FreeSpace / 1MB -as [int]}},
        @{label='Size(GB)';expression={$_.Size / 1GB -as [int]}},
        @{label='%Free';expression={$_.FreeSpace / $_.Size * 100 -as [int]}}
```

← ❶ 参数块

我们只需要在变量声明代码附近添加一个`Param()`块❶。这会将`$computerName`定义为一个参数，并在未对该参数赋值时指定`localhost`作为默认值。你可以不提供默认值，但我们能想到一个合适的值作为默认值时，我们更倾向这么做。

所有以这种方式定义的参数是命名参数，也是位置参数。这意味着我们可以用以下任意一种方式调用该脚本。

```
PS C:\> .\Get-DiskInventory.ps1 server-r2
PS C:\> .\Get-DiskInventory.ps1 -computername server-r2
```

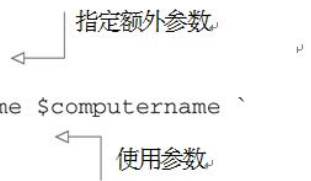
```
PS C:\> .\Get-DiskInventory.ps1 -comp server-r2
```

在第一个实例中，我们以位置参数的形式调用该脚本，只提供参数值而不指定参数名称。在第2、3个实例中，我们指定参数名称，但在第3个实例中，我们将参数名称简化为符合PowerShell的参数名称简化规则的形式。注意，在上面三个示例中，我们都需要为脚本指定路径（.\，也就是当前目录），这是由于Shell并不会搜索当前目录来找到脚本。

你可以通过逗号作为分隔符指定任意数量的参数。例如，假如我们还希望将过滤条件设置为参数。当前脚本仅获取类型为3的驱动器，也就是硬盘。我们可以将该值变为参数，如代码清单21.3所示。

代码清单21.3 Get-DiskInventory.ps1，包含一个额外参数

```
param (
    $computername = 'localhost',
    $drivetype = 3
)
Get-WmiObject -class Win32_LogicalDisk -computername $computername `
    -filter "drivetype=$drivetype" |
Sort-Object -property DeviceID |
Format-Table -property DeviceID,
    @{label='FreeSpace(MB)';expression={$_.FreeSpace / 1MB -as [int]}},
    @{label='Size(GB)';expression={$_.Size / 1GB -as [int]}},
    @{label='%Free';expression={$_.FreeSpace / $_.Size * 100 -as [int]}}
```



注意，我们利用了PowerShell中在双引号中的文本可以自动将变量替换为变量值的功能（你已经在第18章中学到了这个技巧）。

我们可以以最开始的三种方式运行该脚本。当然，我们也可以通过忽略参数的方式使用参数的默认值。下面是一些该脚本的使用示例。

```
PS C:\> .\Get-DiskInventory.ps1 server-r2 3
PS C:\> .\Get-DiskInventory.ps1 -comp server-r2 -drive 3
PS C:\> .\Get-DiskInventory.ps1 server-r2
PS C:\> .\Get-DiskInventory.ps1 -drive 3
```

在第一个示例中，对于两个参数，我们都按照它们在Param()代码块中声明的顺序作为位置参数使用。在第二个示例中，我们对两个参数名称都进行了简化。在第三个示例中，我们完全忽略了-drive参数，从

而使用该参数的默认值3。在最后一个实例中，我们忽略了-computerName，使用该参数的默认值localhost。

21.5 为脚本添加文档

只有真正吝啬的人才会创建一个有用的脚本，而不告诉任何人如何使用它。幸运的是，PowerShell提供了简单的方式为脚本添加帮助，也就是通过注释。你当然可以为你的脚本添加典型编程风格的注释，但如果你已经在脚本中使用了完整的Cmdlet名称和参数名称，很多时候你的脚本的意图已经足够可以望文生义。通过使用特殊的注释语法，你可以提供模仿PowerShell本身帮助文件的帮助信息。

代码清单21.4展示了我们为脚本添加的内容。

代码清单21.4 为Get-DiskInventory.ps1添加帮助

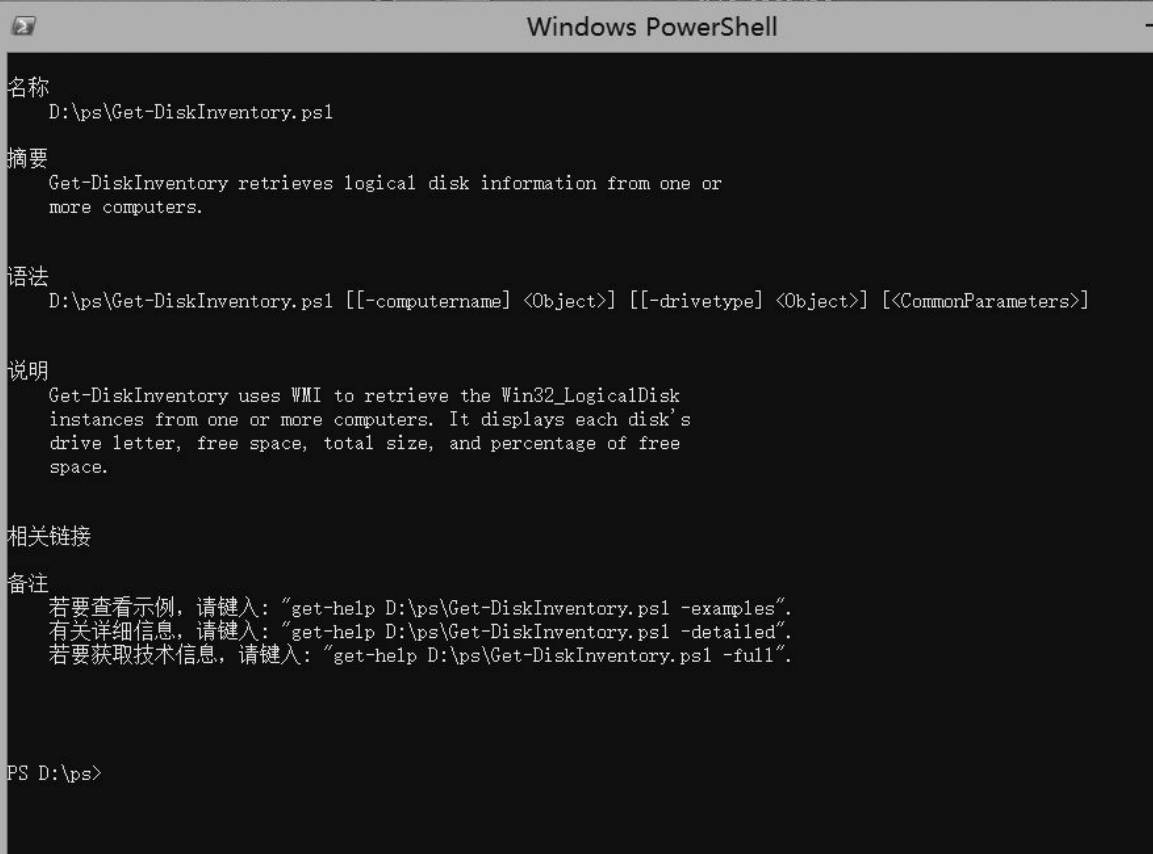
```
<#
.SYNOPSIS
Get-DiskInventory retrieves logical disk information from one or
more computers.
.DESRIPTION
Get-DiskInventory uses WMI to retrieve the Win32_LogicalDisk
instances from one or more computers. It displays each disk's
drive letter, free space, total size, and percentage of free
space.
.PARAMETER computername
The computer name, or names, to query. Default: Localhost.
.PARAMETER drivetype
The drive type to query. See Win32_LogicalDisk documentation
for values. 3 is a fixed disk, and is the default.
.EXAMPLE
Get-DiskInventory -computername SERVER-R2 -drivetype 3
#>
param (
    $computername = 'localhost',
    $drivetype = 3
)
Get-WmiObject -class Win32_LogicalDisk -computername $computername `
    -filter "drivetype=$drivetype" |
Sort-Object -property DeviceID |
Format-Table -property DeviceID,
    @{label='FreeSpace(MB)';expression={$_.FreeSpace / 1MB -as
[int]}},
    @{label='Size(GB)';expression={$_.Size / 1GB -as [int]}},
    @{label='%Free';expression={$_.FreeSpace / $_.Size * 100 -as
```



```
[int]]}
```

正常情况下，PowerShell都会忽略以#开头的代码行，意味着#用于标识某一行是注释。而我们使用<# #>块注释语法，这是由于我们需要注释多行而不希望在每一行开始都使用#。

现在我们可以使用标准的控制台宿主，并通过运行`Help .\Get-DiskInventory`命令获取帮助。（再一次，我们需要提供路径，这是由于该脚本并不是一个内置Cmdlet。）图21.2显示了该命令的输出结果，证明了PowerShell读取并根据这些注释创建了标准的帮助显示界面。我们甚至可以运行`help .\Get-DiskInventory -full`来获取完整的帮助，其中包括了参数信息和示例。图21.3显示了该结果。



```
Windows PowerShell

名称
D:\ps\Get-DiskInventory.ps1

摘要
Get-DiskInventory retrieves logical disk information from one or
more computers.

语法
D:\ps\Get-DiskInventory.ps1 [[-computername] <Object>] [[-drivetype] <Object>] [<CommonParameters>]

说明
Get-DiskInventory uses WMI to retrieve the Win32_LogicalDisk
instances from one or more computers. It displays each disk's
drive letter, free space, total size, and percentage of free
space.

相关链接

备注
若要查看示例，请键入：“get-help D:\ps\Get-DiskInventory.ps1 -examples”。
有关详细信息，请键入：“get-help D:\ps\Get-DiskInventory.ps1 -detailed”。
若要获取技术信息，请键入：“get-help D:\ps\Get-DiskInventory.ps1 -full”。

PS D:\ps>
```

图21.2 通过标准的帮助命令查看帮助

```
名称
D:\ps\Get-DiskInventory.ps1

摘要
Get-DiskInventory retrieves logical disk information from one or
more computers.

语法
D:\ps\Get-DiskInventory.ps1 [[-computername] <Object>] [[-drivetype] <Object>] [<CommonParameters>]

说明
Get-DiskInventory uses WMI to retrieve the Win32_LogicalDisk
instances from one or more computers. It displays each disk's
drive letter, free space, total size, and percentage of free
space.

参数
-computername <Object>
    The computer name, or names, to query. Default: Localhost.

    是否必需?          False
    位置?              1
    默认值              localhost
    是否接受管道输入?   false
    是否接受通配符?     False

-drivetype <Object>
    The drive type to query. See Win32_LogicalDisk documentation
    for values. 3 is a fixed disk, and is the default.

    是否必需?          False
    位置?              2
    默认值              3
    是否接受管道输入?   false
    是否接受通配符?     False

<CommonParameters>
    此 Cmdlet 支持常见参数: Verbose、Debug、
    ErrorAction、ErrorVariable、WarningAction、WarningVariable、
    OutBuffer、PipelineVariable 和 OutVariable。有关详细信息, 请参阅
    about_CommonParameters (http://go.microsoft.com/fwlink/?LinkID=113216)。

输入

输出

-- More --
```

图21.3 基于支持诸如-example、-detailed以及-full的帮助选项

这些特殊的注释被称为基于注释的帮助，必须置于脚本文件的开始部分。除了我们使用的.DESRIPTION和.SYNOPSIS关键字之外，还有一些关键字。在PowerShell中运行help about_comment_based_help查看完整的列表。

21.6 一个脚本，一个管道

我们通常会告诉人们脚本中包含的任何代码和手动输入PowerShell的代码，或是将脚本中的代码通过剪贴板粘贴到Shell中的代码，运行起来并无不同。

但这并不完全正确。

请考虑下面的简单脚本。

```
Get-Process  
Get-Service
```

仅仅是两个命令，但如果我们将这两个命令手动复制到Shell中，每个命令后按回车键执行会发生什么？

动手实验： 你需要自己尝试运行这些命令查看结果；该命令的输出结果过长，以致难以将结果甚至结果截图放入书中。

当你分别运行命令时，你会为每一个命令创建一个新的管道。在每一个管道末尾，PowerShell会查看哪一列需要被格式化并创建一个你可以看到的表格。这里的重点是“不同命令运行在不同管道中”。图21.4阐述了这一点：两个完全分开的命令，两个独立的管道，两个格式化进程，两个不同界面的结果集。

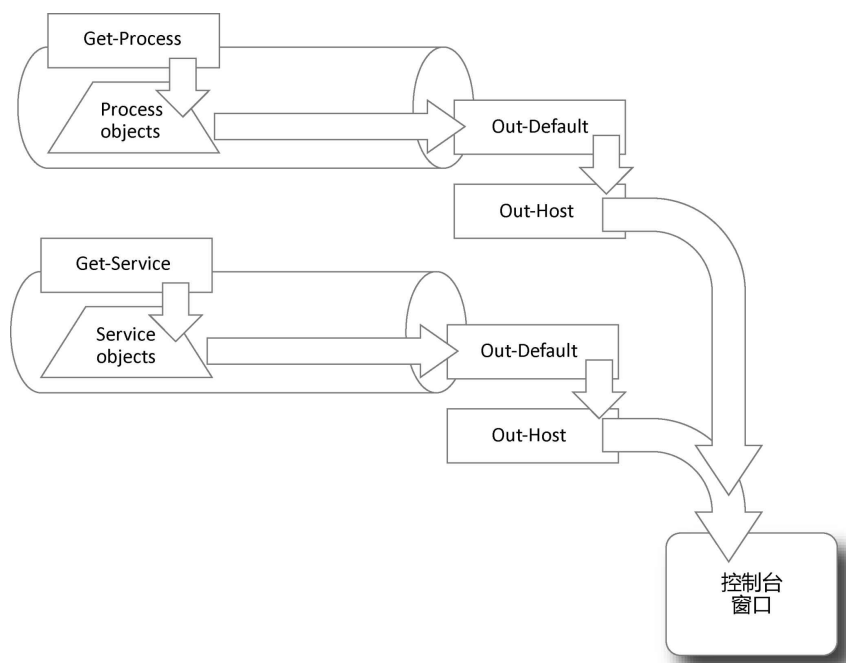


图21.4 两个命令、两个管道、在同一个控制台窗口中的两个输出结果集

你或许会认为我们用了大量篇幅介绍显而易见的内容有些大题小做，但这很重要。下面是分别运行这两个命令经历的步骤：

- (1) 运行**Get-Process**；
- (2) 该命令将**Process**对象放入管道；
- (3) 管道以**Out-Default**结束，该命令会接收对象；
- (4) **Out-Default**将对象传递给**Out-Host**，该命令会调用格式化系统产生文本输出结果（你在第10章学到过这些）；
- (5) 文本输出结果显示在屏幕上；
- (6) 运行**Get-Service**；
- (7) 该命令将**Service**对象放入管道；
- (8) 管道以**Out-Default**结束，该命令会接收对象；
- (9) **Out-Default**将对象传递给**Out-Host**，该命令会调用格式化系统产生文本输出结果；
- (10) 文本输出结果显示在屏幕上。

所以你现在看到屏幕包含了来自两个命令的结果。我们希望你将这两个命令放入脚本文件，并命名为**Test.ps1**或其他简单的名称。在运行脚本之前，将这两个命令复制到剪贴板，你可以选中这两行并按**Ctrl+C**组合键将其复制到剪贴板。

转到**PowerShell**控制台宿主并按下回车键。这会将剪贴板中的命令粘贴到**Shell**中。在**Shell**中执行的方式会和**ISE**中完全一致，这是由于回车也会被粘贴进来。再一次，你在两个管道中运行不同的命令。

现在回到**ISE**中并运行脚本，结果不同，对吧？这是什么原因？

在**PowerShell**中，所有的命令都在一个管道中执行，在脚本中也是同样。在脚本中，任何产生管道输出结果的命令都会被写入同一个管道中：脚本自身运行的管道。请查看图21.5。

我们尝试解释发生了什么：

- (1) 脚本运行**Get-Process**。
- (2) 该命令将**Process**对象放入管道。
- (3) 脚本运行**Get-Service**。
- (4) 该命令将**Service**对象放入管道。
- (5) 管道以**Out-Default**结束，该命令会接收上面两类对象。

(6) **Out-Default**将对象传递给**Out-Host**，该命令会调用格式化系统产生文本输出结果。

(7) 由于**Process**对象首先被放入管道，**Shell**的格式化系统会为**Process**对象选择合适的格式化方式。这也是为什么**Process**对象的输出结果看起来很正常。当**Shell**碰到**Service**对象后，它会生成一个全新的表，所以会最终生成一个列表。

- (8) 屏幕显示文本输出结果。

两种不同的输出是由于将两种类别的对象放入一个管道中。这是将命令存入脚本和手动执行之间的重要区别：在脚本中，只能够使用一个管道。正常来讲，你的脚本应该努力保持只输出一类对象，以便**PowerShell**能产生合理的文本输出格式。

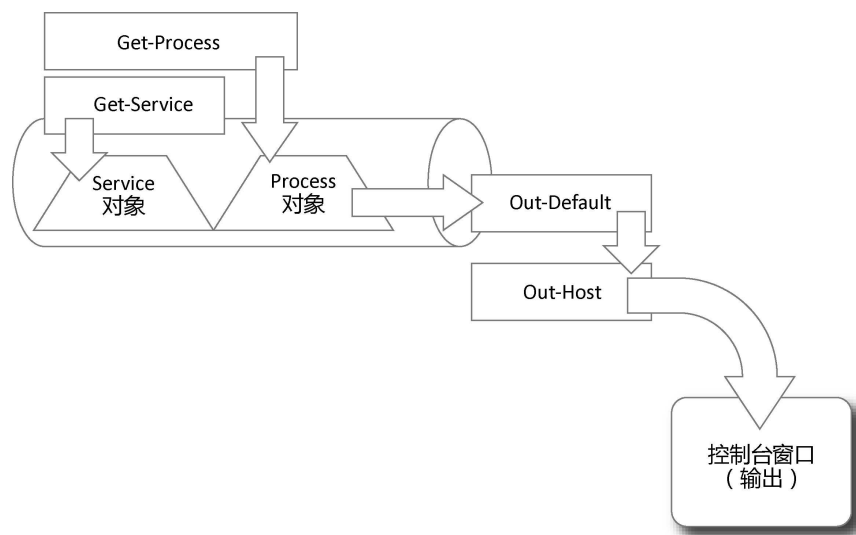


图21.5 在一个脚本中，所有的命令都是在该脚本单独的管道中执行

21.7 作用域初探

我们最后需要讨论的一个主题是作用域（scope）。作用域是特定类型PowerShell元素的容器，这些元素主要是别名、变量和函数。

Shell本身具有最高级的作用域，称为全局域（global scope）。当运行一个脚本时，会在脚本范围内创建一个新的作用域，也就是所谓的脚本作用域（script scope）。脚本作用域是全局作用域的子集，也就是全局作用域的子作用域（child）。而全局作用域是脚本作用域的父作用域（parent）。函数还有其特有的私有作用域（private scope）。

图21.6描述了这些作用域之间的关系，全局作用域包含了其子作用域，而其子作用域包含了其他子作用域，以此类推。

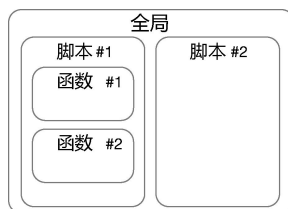


图21.6 全局脚本及函数（私有）作用域

作用域的生命周期只持续到作用域所需执行的最后一行代码之前。这意味着全局作用域只有在PowerShell运行时有效，脚本作用域只在脚本运行时有效，以此类推。一旦停止运行，作用域和其包含的内容同时消失。PowerShell对于别名、变量和函数之类的元素有着非常详细——某些时候也是非常让人困惑的规则，但主要规则是，如果你尝试访问一个作用域元素，PowerShell在当前作用域内查找，如果不存在于当前作用域，PowerShell会查找其父作用域，以此类推，直到找到树形关系的顶端——也就是全局作用域。

动手实验： 为了获得正确的结果，请小心按照下面的指导操作，这非常重要。

让我们进行实战，遵循下面的步骤。

(1) 关闭已经打开的PowerShell或PowerShell ISE窗口，这样你就可以从头开始。

(2) 打开一个新的PowerShell或PowerShell ISE窗口。

(3) 在ISE中，创建一个包含一行命令的脚本，该命令为Write \$x。

(4) 将脚本保存到c:\scope.ps1。

(5) 在一个标准的PowerShell窗口，使用命令C:\Scope运行脚本。没有任何输出结果。当脚本运行时，会自动为其创建一个新的作用域。而\$x变量在该作用域内并不存在，因此PowerShell转向其父作用域——也就是全局作用域检查变量\$x是否存在。该变量在父作用域也不存在，因此PowerShell认为\$x为空，并打印出空（也就是不输出任何结果）作为输出结果。

(6) 在一个标准的PowerShell窗口，运行\$x = 4，然后再次运行C:\Scope。这次，你会按到输出结果为4。虽然变量\$x在脚本范围内未定义，但PowerShell可以在全局作用域内找到该变量。因此脚本可以使用全局作用域内的值。

(7) 在ISE中，在脚本的开始添加\$x=10（也就是write命令之前），并保存脚本。

(8) 在标准的PowerShell窗口中，再次运行C:\Scope。这次，你会看到输出结果为10。这是由于\$x在脚本作用域内定义，因此Shell无须查看全局作用域。现在在Shell中运行\$x。你将看到输出结果为4，这意味着在脚本作用域内的变量值不会影响全局作用域内的变量值。

在这里一个重要的概念是，当在作用域内定义一个变量、别名或函数时，当前作用域就无法访问父作用域内的任何同名变量、别名或函数。PowerShell总会使用局部定义的元素。例如，如果你将New-Alias Dir Get-Service命令放入一个脚本，那么在当前脚本中，别名Dir总是运行Get-Service而不是Get-ChildItem（实际上，Shell很可能不允许你这么做，这是由于其需要保护内置别名不会重新被定义）。通过在脚本作用域内定义别名，你可以防止Shell去父作用域查找标准和默认的Dir。当然，对于Dir别名的重定义只能持续到脚本执行结束之前，而全局作用域默认的Dir将不受影响。

这些作用域相关的理念可能会让你感到困惑。你可以通过永远不依赖除了当前作用域内的其他作用域来避免这种混淆。因此在尝试在脚本中访问一个变量时，请确保你已经在同一个作用域内给其赋值。在

Param()块内的参数可以实现这一点，还有很多其他方式可以将值或对象赋予一个变量。

21.8 动手实验

注意： 对于本次动手实验来说，你需要运行PowerShell v3或更新版本PowerShell的计算机。

将下面的命令添加到一个脚本中。你首先需要识别出需要定义为参数的元素，比如说计算机名称。最终的脚本应该定义好参数，并且你还需要为脚本创建基于注释的帮助。运行脚本从而对脚本进行测试，并使用Help命令，从而确保基于注释的帮助可以正常工作。请不要忘记阅读本章提到的帮助文件以获取更多信息。

下面是命令：

```
Get-WmiObject Win32_LogicalDisk -comp "localhost" -filter  
"drivetype=3" |  
Where { $_.FreeSpace / $_.Size -lt .1 } |  
Select -Property DeviceID,FreeSpace,Size
```

提示如下：你至少可以发现2处信息需要变为参数。该命令用于列出少于给定可用空间的驱动器。显而易见，你并不只想把本地主机作为目标，并且你不希望10%（也就是1）作为阈值。你还可以选择将驱动器类型作为参数（这里也就是3），但是对于动手实验来说，保留其值为3即可。

第22章 优化可传参脚本

在之前章节，我们给你留下了许多非常酷的可传参的脚本。可传参脚本的思想是脚本的使用者无须关心或者干预脚本的内容。他们只需要通过设计好的界面提供输入——也就是参数，使用者能够修改的地方只有参数。在本章，我们将对此更进一步。

22.1 起点

为了确保我们在同一起点，让我们使用代码清单22.1作为起点。该脚本以基于注释的帮助为特点，两个输入参数和一个使用输入参数的命令。我们基于之前章节做了小幅修改：我们将输出结果输出为被选择的对象，而不是格式化之后的表格。

代码清单22.1 起点：Get-DiskInventory.ps1

```
<#
.SYNOPSIS
Get-DiskInventory retrieves logical disk information from one or
more computers.
.DESCRIPTION
Get-DiskInventory uses WMI to retrieve the Win32_LogicalDisk
instances from one or more computers. It displays each disk's
drive letter, free space, total size, and percentage of free
space.
.PARAMETER computername
The computer name, or names, to query. Default: Localhost.
.PARAMETER drivetype
The drive type to query. See Win32_LogicalDisk documentation
for values. 3 is a fixed disk, and is the default.
.EXAMPLE
Get-DiskInventory -computername SERVER-R2 -drivetype 3
#>
param (
    $computername = 'localhost',
    $drivetype = 3
)
Get-WmiObject -class Win32_LogicalDisk -computername $computername
`
```

```
-filter "drivetype=$drivetype" |  
Sort-Object -property DeviceID |  
Select-Object -property DeviceID,  
    @{{name='FreeSpace(MB)';expression={$_.FreeSpace / 1MB -as  
[int]}}},  
    @{{name='Size(GB)';expression={$_.Size / 1GB -as [int]}}},  
    @{{name='%Free';expression={$_.FreeSpace / $_.Size * 100 -as  
[int]}}}
```

为什么我们使用**Select-Object**而不是**Format-Table**？因为我们通常会感觉写一个会产生格式化后的输出结果的脚本不是一个好主意。毕竟，如果某个用户需要**CSV**格式的文件，而脚本输出格式化后的表，该用户就无法完成工作。通过本次修改，我们可以通过下述方式获得格式化后的表。

```
PS C:\> .\Get-DiskInventory | Format-Table
```

或者通过下述方式运行获取**CSV**文件。

```
PS C:\> .\Get-DiskInventory | Export-CSV disks.csv
```

关键点是输出对象（也就是**Select-Object**完成的工作），与格式化的显示结果相反，将会使得我们的脚本从长远角度来说更加灵活。

22.2 让PowerShell去做最难的工作

我们只需在上述脚本的基础上再多加一行脚本来展现**PowerShell**的奇妙。这使得从技术角度来说，把我们的脚本变为所谓的“高级脚本”，使得大量**PowerShell**能做的事得以展现。代码清单22.2展现了修订后的脚本——我们已经对新增行加粗。

代码清单22.2 将Get-DiskInventory.ps1变为高级脚本

```
<#  
.SYNOPSIS
```

```

Get-DiskInventory retrieves logical disk information from one or
more computers.
.DESCRIPTION
Get-DiskInventory uses WMI to retrieve the Win32_LogicalDisk
instances from one or more computers. It displays each disk's
drive letter, free space, total size, and percentage of free
space.
.PARAMETER computername
The computer name, or names, to query. Default: Localhost.
.PARAMETER drivetype
The drive type to query. See Win32_LogicalDisk documentation
for values. 3 is a fixed disk, and is the default.
.EXAMPLE
Get-DiskInventory -computername SERVER-R2 -drivetype 3
#>
[CmdletBinding()]

param (
    $computername = 'localhost',
    $drivetype = 3
)
Get-WmiObject -class Win32_LogicalDisk -computername $computername `
    -filter "drivetype=$drivetype" |
Sort-Object -property DeviceID |
Select-Object -property DeviceID,
    @{name='FreeSpace(MB)';expression={$_.FreeSpace / 1MB -as
[int]}},
    @{name='Size(GB)';expression={$_.Size / 1GB -as [int]}},
    @{name='%Free';expression={$_.FreeSpace / $_.Size * 100 -as
[int]}}

```

在基于备注的帮助代码后面，将[CmdletBinding()]指示符置于脚本的第一行非常重要。PowerShell只会在该位置查看该指示符。加上这个指示符之后，脚本还会正常运行。但我们已经启用了好几个功能，我们会在接下来进行探索。

22.3 将参数定义为强制化参数

我们对现有的脚本并不满意，这是由于它提供了默认的-ComputerName参数。我们并不确定真正需要该参数。我们更倾向于选择提示用户输入值。幸运的是，PowerShell中实现该功能很简单——同样，只需要添加一行代码就能完成，如代码清单22.3所示。

代码清单22.3 为Get-DiskInventory.ps1添加一个强制参数

```
<#
.SYNOPSIS
Get-DiskInventory retrieves logical disk information from one or
more computers.
.DESCRIPTION
Get-DiskInventory uses WMI to retrieve the Win32_LogicalDisk
instances from one or more computers. It displays each disk's
drive letter, free space, total size, and percentage of free
space.
.PARAMETER computername
The computer name, or names, to query. Default: Localhost.
.PARAMETER drivetype
The drive type to query. See Win32_LogicalDisk documentation
for values. 3 is a fixed disk, and is the default.
.EXAMPLE
Get-DiskInventory -computername SERVER-R2 -drivetype 3
#>
[CmdletBinding()]
param (
    [Parameter(Mandatory=$True)]
    [string]$computername,

    [int]$drivetype = 3
)
Get-WmiObject -class Win32_LogicalDisk -computername $computername `
    -filter "drivetype=$drivetype" |
    Sort-Object -property DeviceID |
    Select-Object -property DeviceID,
        @{name='FreeSpace(MB)';expression={$_.FreeSpace / 1MB -as
[int]}},
        @{name='Size(GB)';expression={$_.Size / 1GB -as [int]}},
        @{name='%Free';expression={$_.FreeSpace / $_.Size * 100 -as
[int]}}
```

补充说明

当某个用户使用你写的脚本却没有为强制参数提供值时，PowerShell将会提示他输入。有两种方式可以使得PowerShell给用户提供有意义的提示。

首先，使用有意义的参数名称。提示用户为名称为“comp”的参数赋值，远不如提示用户为名称为“ComputerName”的参数赋值有意义。所以请尝试使用具有自描述性的参数名称，并与其他PowerShell命令使用的参数名称保持一致。

你还可以添加一条帮助信息：

```
[Parameter(Mandatory=$True, HelpMessage="Enter a computer name to query")]
```

某些PowerShell宿主程序将会将帮助信息作为提示的一部分，使得用户获得更简洁的帮助信息。但并不是所有的宿主应用程序都会使用该标签，所以你测试的时候没有看到提示的帮助信息也不用沮丧。当我们写一些给他人使用的脚本时，我们喜欢在脚本中将帮助信息包含在内。这么做永远不会有任何坏处。但是为了简便起见，我们不会在本章的示例中添加帮助信息。

仅仅使用[Parameter(Mandatory=\$True)]这样一个描述符，会使得当用户忘记提供计算机名称时，PowerShell就会提示用户输入该参数。为了更进一步帮助PowerShell识别用户传入的参数，我们定义两个输入参数的数据类型：-computerName定义为[string]类型，而-drivetype定义为INT（也就是整型）。

将这类标签添加到参数会让人困惑，因此让我们更进一步查看Param()代码块的语法，如图22.1所示。

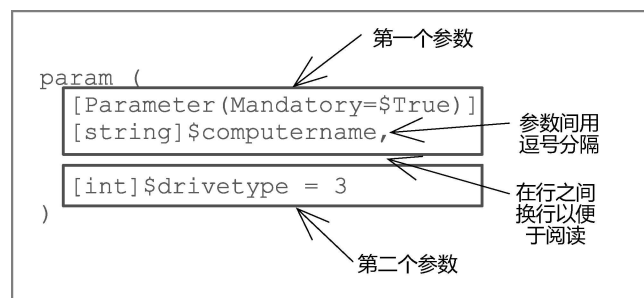


图22.1 分解Param()代码段的语法

下面是需要注意的重点。

- 所有的参数都必须被包括在Param()代码段的括号内。
- 可以对一个参数添加多个修饰符，多个修饰符既可以是一行，也可以是图22.1中那样的多行。我们认为多行更易于阅读，但重点是即使是多行，它们也是一个整体。Mandatory标签仅修改-computerName——它对-drivetype并没有影响。
- 除了最后一个参数之外，所有的参数之间以逗号分隔。
- 为了更好的可读性，我们还喜欢在参数之间添加空格。我们认为空格会使得从视觉上分隔参数更加容易，从而减少Param()代码段引起的困惑。
- 我们在定义参数时，就好像参数是变量——\$computername和\$drivetype——但使用该脚本的人会将其当作普通的PowerShell命令行参数，比如说-computername和-drivetype。

动手实验： 将代码清单22.3中的脚本保存，并在Shell中运行。不要为-computername参数赋值，从而可以查看PowerShell是如何提示你输入该参数的。

22.4 添加参数别名

当你想到计算机名称时，“computername”是否是你想到的第一个词？或许不是。我们使用-computerName作为参数名称，是因为该参数名称与其他PowerShell命令一致。查看Get-Service、Get-WmiObject、Get-Process以及其他命令，你可以发现这些命令都使用-computerName作为参数名称。所以我们也同样使用该名称作为参数名称。

但假如你认为-hostname更容易记忆的话，你可以将该名称作为备用名称添加，也就是参数名别。只需要另外一个修饰符，如代码清单22.4所示。

代码清单22.4 为Get-DiskInventory.ps1添加一个别名

```
<#
.SYNOPSIS
Get-DiskInventory retrieves logical disk information from one or
more computers.
.DESRIPTION
Get-DiskInventory uses WMI to retrieve the Win32_LogicalDisk
instances from one or more computers. It displays each disk's
```

```

drive letter, free space, total size, and percentage of free
space.
.PARAMETER computername
The computer name, or names, to query. Default: Localhost.
.PARAMETER drivetype
The drive type to query. See Win32_LogicalDisk documentation
for values. 3 is a fixed disk, and is the default.
.EXAMPLE
Get-DiskInventory -computername SERVER-R2 -drivetype 3
#>
[CmdletBinding()]
param (
    [Parameter(Mandatory=$True)]
    [Alias('hostname')]
    [string]$computername,

    [int]$drivetype = 3
)
Get-WmiObject -class Win32_LogicalDisk -computername $computername
-
-filter "drivetype=$drivetype" |
Sort-Object -property DeviceID |
Select-Object -property DeviceID,
    @{name='FreeSpace(MB)';expression={$_.FreeSpace / 1MB -as
[int]}},
    @{name='Size(GB)';expression={$_.Size / 1GB -as [int]}},
    @{name='%Free';expression={$_.FreeSpace / $_.Size * 100 -as
[int]}}

```

完成小幅修改后，我们现在可以运行下述代码。

```
PS C:\> .\Get-DiskInventory -host SERVER2
```

注意： 请记住，你只需输入足够让PowerShell分辨出是哪个参数的部分参数名即可。在本例中，-host足以让PowerShell识别出指的是-hostname参数。当然，我们也可以输入完整的参数名称。

再次声明，新增的标签是-computerName参数的一部分，因此对-drivetype参数不生效。现在-computerName参数的定义占用了三行。当然，我们也能将三行连成一行：

```
[Parameter(Mandatory=$True)][Alias('hostname')]
```

```
[string]$computername,
```

我们只是认为这种方式更加难以阅读。

22.5 验证输入的参数

让我们和`-drivetype`参数打打交道。根据MSDN中`Win32_LogicalDisk`这个WMI类的文档（搜索类名称，在结果中，前几条记录中就有该文档），驱动器类型3是本地磁盘。类型2是可移动磁盘。可移动磁盘也会计算容量以及可用空间。驱动类型1、4、5、6更少被使用（还有人在继续使用类型6的RAM驱动器吗？），在某些情况下，有一些磁盘没有可用空间（比如类型为5的光盘）。所以我们希望阻止使用我们脚本的用户使用这些类型。

代码清单22.5展示了我们所需做的小幅修改。

代码清单22.5 为Get-DiskInventory.ps1添加参数验证

```
<#
.SYNOPSIS
Get-DiskInventory retrieves logical disk information from one or
more computers.
.DESCRIPTION
Get-DiskInventory uses WMI to retrieve the Win32_LogicalDisk
instances from one or more computers. It displays each disk's
drive letter, free space, total size, and percentage of free
space.
.PARAMETER computername
The computer name, or names, to query. Default: Localhost.
.PARAMETER drivetype
The drive type to query. See Win32_LogicalDisk documentation
for values. 3 is a fixed disk, and is the default.
.EXAMPLE
Get-DiskInventory -computername SERVER-R2 -drivetype 3
#>
[CmdletBinding()]
param (
    [Parameter(Mandatory=$True)]
    [Alias('hostname')]
    [string]$computername,
```



```
[ValidateSet(2, 3)]
[int]$drivetype = 3

)
Get-WmiObject -class Win32_LogicalDisk -computername $computername `
-filter "drivetype=$drivetype" |
Sort-Object -property DeviceID |
Select-Object -property DeviceID,
    @{name='FreeSpace(MB)';expression={$_.FreeSpace / 1MB -as
[int]}},
    @{name='Size(GB)';expression={$_.Size / 1GB -as [int]}},
    @{name='%Free';expression={$_.FreeSpace / $_.Size * 100 -as
[int]}}
```

新的标签告诉PowerShell，对于参数-drivetype，只允许传入值2和3，并且3是默认值。

还有一系列其他可以添加到参数的验证技术。当这样做有意义时，可以将多个修饰符添加到同一个参数上。运行**help about_functions_advanced_parameters**可以获得完整列表——目前为止，我们只使用ValidateSet。Jeffery还写了一个关于其他可能用上的“验证”标签的系列博客——你可以在网站<http://jdhitsolutions.com/blog/>上查看到该系列博客（搜索“validate”）。

动手实验： 将这段代码保存并再次运行——尝试指定-drivetype参数为5，看看PowerShell是如何响应的。

22.6 通过添加详细输出获得用户友好体验

在第19章中，我们提到，我们倾向于使用Write-Verbose而不是Write-Host来产生一些人喜欢看到脚本产生的逐步进度信息。下面让我们来看一个实际例子。

我们在代码清单22.6中添加一些详细输出。

代码清单22.6 为Get-DiskInventory.ps1添加详细输出

```
<#
```

```
.SYNOPSIS
Get-DiskInventory retrieves logical disk information from one or
more computers.
.DESCRIPTION
Get-DiskInventory uses WMI to retrieve the Win32_LogicalDisk
instances from one or more computers. It displays each disk's
drive letter, free space, total size, and percentage of free
space.
.PARAMETER computername
The computer name, or names, to query. Default: Localhost.
.PARAMETER drivetype
The drive type to query. See Win32_LogicalDisk documentation
for values. 3 is a fixed disk, and is the default.
.EXAMPLE
Get-DiskInventory -computername SERVER-R2 -drivetype 3
#>
[CmdletBinding()]
param (
    [Parameter(Mandatory=$True)]
    [Alias('hostname')]
    [string]$computername,

    [ValidateSet(2, 3)]
    [int]$drivetype = 3
)
Write-Verbose "Connecting to $computername"
Write-Verbose "Looking for drive type $drivetype"
Get-WmiObject -class Win32_LogicalDisk -computername $computername `
    -filter "drivetype=$drivetype" |
Sort-Object -property DeviceID |
Select-Object -property DeviceID,
    @{name='FreeSpace(MB)';expression={$_.FreeSpace / 1MB -as
[int]}},
    @{name='Size(GB)';expression={$_.Size / 1GB -as [int]}},
    @{name='%Free';expression={$_.FreeSpace / $_.Size * 100 -as
[int]}}
Write-Verbose "Finished running command"
```

下面尝试以两种方式运行该脚本。第一次尝试不会显示任何详细输出。

```
PS C:\> .\Get-DiskInventory -computername localhost
```

下面是第二次尝试，也就是我们希望显示详细输出。

```
PS C:\> .\Get-DiskInventory -computername localhost -verbose
```

动手实验：当你自己动手尝试时就会发现酷很多——尝试运行我们展示的脚本，并查看两次运行的差别。

太酷了，不是吗？当你想要详细输出时，就能获得详细输出——并且完全无须为**-Verbose**参数赋值。当添加[**CmdletBinding()**]时，就可以无成本拥有详细输出。最妙的部分是，该标签还会激活脚本中所包含命令的详细输出！所以你使用的任何被设计可以产生详细输出结果的命令都会自动输出详细结果。该技术使得启用或禁用详细输出变得非常容易，相比**Write-Host**更加灵活。而且你无须通过操作**\$VerbosePreference**变量，使得结果展现在屏幕上。

同时，注意在详细输出中我们是如何使用PowerShell的双引号技巧的：通过将变量（**\$computername**）包含在双引号中，输出内容就可以包含变量的内容，所以我们可以看到PowerShell输出该变量的内容。

22.7 动手实验

注意：对于本次动手实验来说，你需要运行PowerShell v3或更新版本PowerShell的计算机。

本次动手实验需要你回忆起在第12章所学内容，因为你需要将下述命令参数化，并将其存入脚本——正如你在第21章所做的那样。但这次我们还需要你将**-ComputerName**参数变为强制参数，并给它一个名称为**hostname**的别名。并且使得你的脚本可以在运行命令之前和之后显示详细输出。请记住，你必须将计算机名称参数化——这也是在本次案例中你唯一需要参数化的参数。

请确保在修改之前运行下述命令，从而确保下述命令可以在你的系统上运行。

```
get-wmiobject win32_networkadapter -computername localhost |  
where { $_.PhysicalAdapter } |  
select MACAddress,AdapterType,DeviceID,Name,Speed
```

重申一下，这里是你需要完成的任务列表。

- 确保该命令在修改之前可以正常运行。
- 将计算机名称参数化。
- 将-**ComputerName**参数变为强制参数。
- 给予计算机名称参数一个别名**hostname**。
- 至少添加一个如何使用本脚本的基于注释的帮助。
- 在命令运行之前和之后添加详细输出结果。
- 将脚本保存为**Get-PhysicalAdapters.ps1**。

第23章 高级远程配置

在第13章中，我们尽最大努力为你介绍PowerShell的远程技术。我们故意留下一些硬骨头，从而使得我们可以专注于远程背后的核心技术。但是在本章，我们希望重新提起这些硬骨头，并阐述一些更加高级和不常用的功能与场景。我们必须提前承认并不是本章所有的内容都能够派上用场——但是我们认为每个人都应该了解这些选项，以防之后对这些选项有需求。

同时，我们提醒你，本书主要内容是关于PowerShell v3以及之后版本。关于找出当前运行的版本的办法，请重新查看第1章。本书涵盖的大部分内容无法运行在之前版本中。

23.1 使用其他端点

正如你在第13章中所学那样，一台计算机可以包含多个端点。在PowerShell中，端点也被称为会话配置（session configurations）。举例来说，在64位机器上启用远程会同时为32位PowerShell和64位PowerShell各启用一个端点，其中64位PowerShell的端点是默认端点。

如果你拥有管理员权限，你可以在任何计算机下运行下述命令，获得可用的会话配置列表。

```
PS C:\> Get-PSSessionConfiguration

Name           : microsoft.powerShell
PSVersion      : 3.0
StartupScript  :
RunAsUser      :
Permission     : NT AUTHORITY\NETWORK AccessDenied,
                BUILTIN\Administrators
                AccessAllowed
Name           : microsoft.powerShell.workflow
PSVersion      : 3.0
StartupScript  :
RunAsUser      :
Permission     : NT AUTHORITY\NETWORK AccessDenied,
```

```
BUILTIN\Administrators
    AccessAllowed
Name           : microsoft.PowerShell32
PSVersion      : 3.0
StartupScript  :
RunAsUser       :
Permission     : NT AUTHORITY\NETWORK AccessDenied,
BUILTIN\Administrators
    AccessAllowed
```

每一个端点有一个名称；其中一个名称为“Microsoft.PowerShell”的端点是那些诸如New-PSSession、Enter-PSSession、Invoke-Command等远程命令默认使用的端点。在64位系统中，端点是64位的Shell；在32位系统中，“Microsoft.PowerShell”是32位的Shell。

你可以注意到，我们的64位系统有一个运行32位Shell的备用端点：“Microsoft.PowerShell32”用于兼容性目的。如果希望连接到备用端点，只需要在远程命令的-ConfigurationName参数中指定端点名称。

```
PS C:\> Enter-PSSession -ComputerName DONJONES1D96 -
ConfigurationName 'Microsoft.PowerShell32'
[DONJONES1D96]: PS C:\Users\donjones\Documents>
```

什么时候你会使用备用端点？当你需要运行的命令依赖于32位的PowerShell插件时，或许就是你需要显式通过32位的端点连接到64位的机器上的原因。可能还存在自定义端点。当你需要执行一些特定任务时，你或许需要连接到这些端点上。

23.2 创建自定义端点

创建一个自定义端点可以分为以下两步。

(1) 通过New-PSSessionConfigurationFile命令创建一个新的会话配置文件，该文件的扩展名为PSSC。该文件用于定义端点的特征。特征主要指的是该端点允许运行的命令和功能。

(2) 通过**Register-PSSessionConfiguration**命令载入.PSSC文件，并在WinRm服务中创建新的端点。在注册过程中，你可以设置多个可选参数，比如说谁可以连接到端点。你也可以在必要时通过**Set-PSSessionConfiguration**命令改变设置。

我们将会带领你经历一个使用自定义端点进行授权管理的示例，这或许是PowerShell最酷的功能之一。我们可以创建一个只有域中HelpDesk组的成员可以访问的端点。在端点内，我们启用与网络适配器相关的命令——并且只允许这些命令。我们并不打算给HelpDesk组运行命令的权限，仅仅是让他们可以看到命令。我们还配置端点在我们提供的备用凭据下运行命令，因此可以使得HelpDesk组可以在本身无须拥有执行命令的权限时执行命令。

23.2.1 创建会话配置

下面是我们运行的命令——我们将该命令格式化以便于阅读，但实际上，我们输入后只有一行。

```
PS C:\> New-PSSessionConfigurationFile
-Path C:\HelpDeskEndpoint.pssc
-ModulesToImport NetAdapter
-SessionType RestrictedRemoteServer
-CompanyName "Our Company"
-Author "Don Jones"
-Description "Net adapter commands for use by help desk"
-PowerShellVersion '3.0'
```

这里有一些关键参数，我们已经用粗体重点标注。我们将会解释为什么我们赋了这些值。我们将阅读帮助找出这些参数其他选项的任务留给你。

- **-Path** ;参数是必需的，并且你提供的文件名称必须以.pssc结尾。
- **-ModulesToImport** ;列出组件（在本例中，只有一个名称为NetAdapter的组件），我们只希望对于本端点只有该组件可用。
- **-SessionType RestrictedRemoteServer** ;除了一些必需的命令，移除所有PowerShell的核心命令。该列表会很小，包括Select-Object、Measure-Object、Get-Command、Get-Help、Exit-PSSession等。

- **-PowerShellVersion** ;默认为3.0。在本例中，我们将该参数包含在内，只是为了完整性。

还有一些以**-Visible**开头的参数，比如说**-VisibleCmdlets**。正常情况下，当你使用**-ModulesToImport**导入一个组件时，所有该组件中的命令都会对于使用最终端点地人可见。通过只列出你希望人们看到的**Cmdlet**、别名、函数、提供程序，你非常有效地隐藏了其他内容。这是限制人们通过该端口所能做的操作的好办法。请小心使用**visibility**参数，这是因为该参数有一点让人迷惑。举例来说，如果你导入由**Cmdlet**和函数组成的组件，使用**VisibleCmdlets**仅仅限制能够显示的**Cmdlets**——对于是否显示函数却毫无影响，这意味着这些函数在默认情况下都会被启用。

注意，没有任何方法可以对用户使用的参数进行限制：**PowerShell**支持参数级别的限制，但需要在**Visual Studio**中进行大量编码。这超出了本书的内容。还有你可以使用的其他高级技巧，比如说创建用于隐藏参数的代理函数。但这超出本书的篇幅，因为本书的目标读者是初学者。

23.2.2 会话注册

完成会话配置文件的创建之后，可以通过下述命令使其生效。我们再一次将代码格式化以便于阅读，但实际上只有很长的一行。

```
PS C:\> Register-PSSessionConfiguration  
-Path .\HelpDeskEndpoint.pssc  
-RunAsCredential COMPANY\HelpDeskProxyAdmin  
-ShowSecurityDescriptorUI  
-Name HelpDesk
```

这就创建了名称为**HelpDesk**的新端点。如图23.1所示，提示我们输入**COMPANY\ HelpDeskProxyAdmin**账户的密码；该端点运行的所有命令都通过该账户的身份运行，我们需要确保该账户拥有运行网络适配器相关的命令的权限。

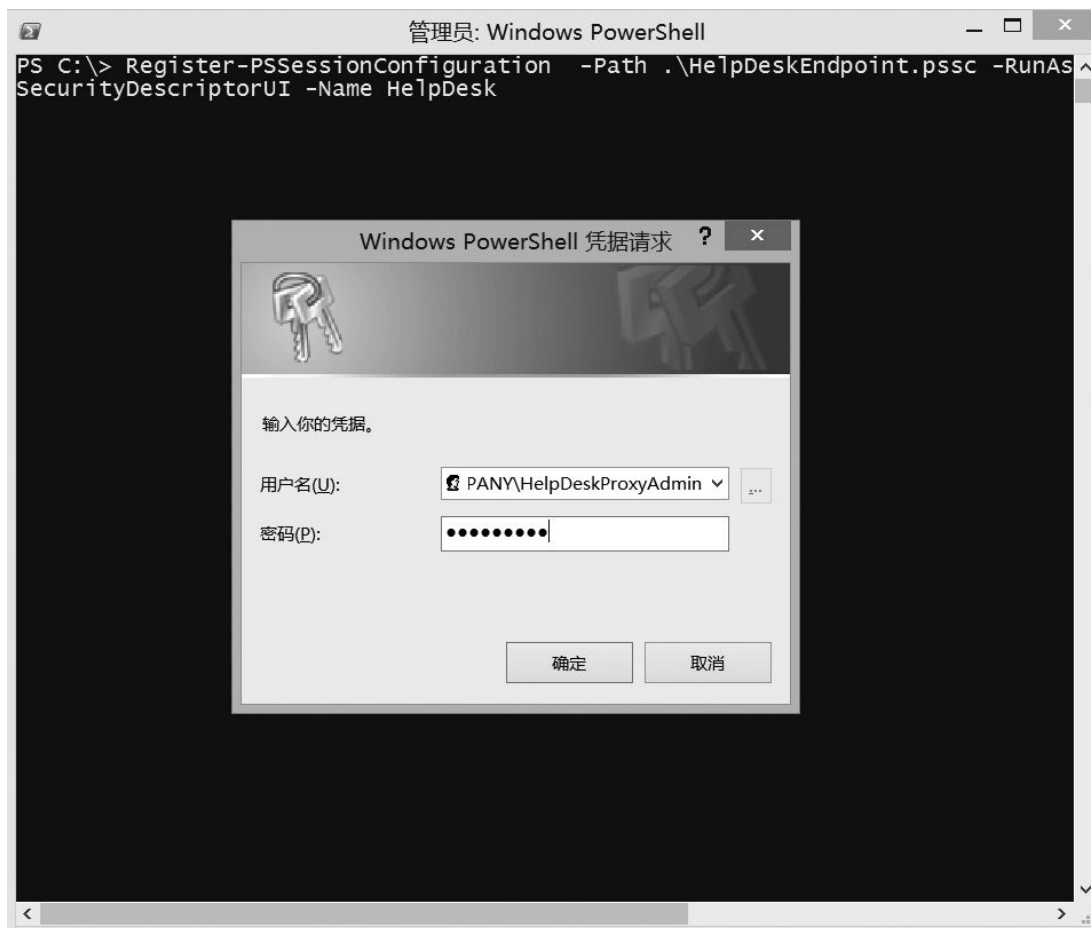


图23.1 提示输入以凭据运行的密码

我们完成几个“是否继续运行”的提示，建议你仔细阅读提示。该命令会停止并重启WinRM服务，这会导致中断其他管理员管理本地机器，所以请小心。

如图23.2所示，还为我们提供了图形化对话框指定哪个用户可以连接到端点。之所以会显示对话框，是由于我们使用了-ShowSecurityDescriptorUI参数，而不是使用复杂的安全描述符定义语言（SDDL）设置权限。坦白讲，这也是我们不熟悉的语言。这同时是相对于Shell使用GUI方式更好的例子——我们将HelpDesk用户组添加在内，并确保该组拥有执行和读权限。执行是所需的最小权限，执行权限将我们计划给该账号的权限赋予端点；读权限是另一个我们需要的权限。

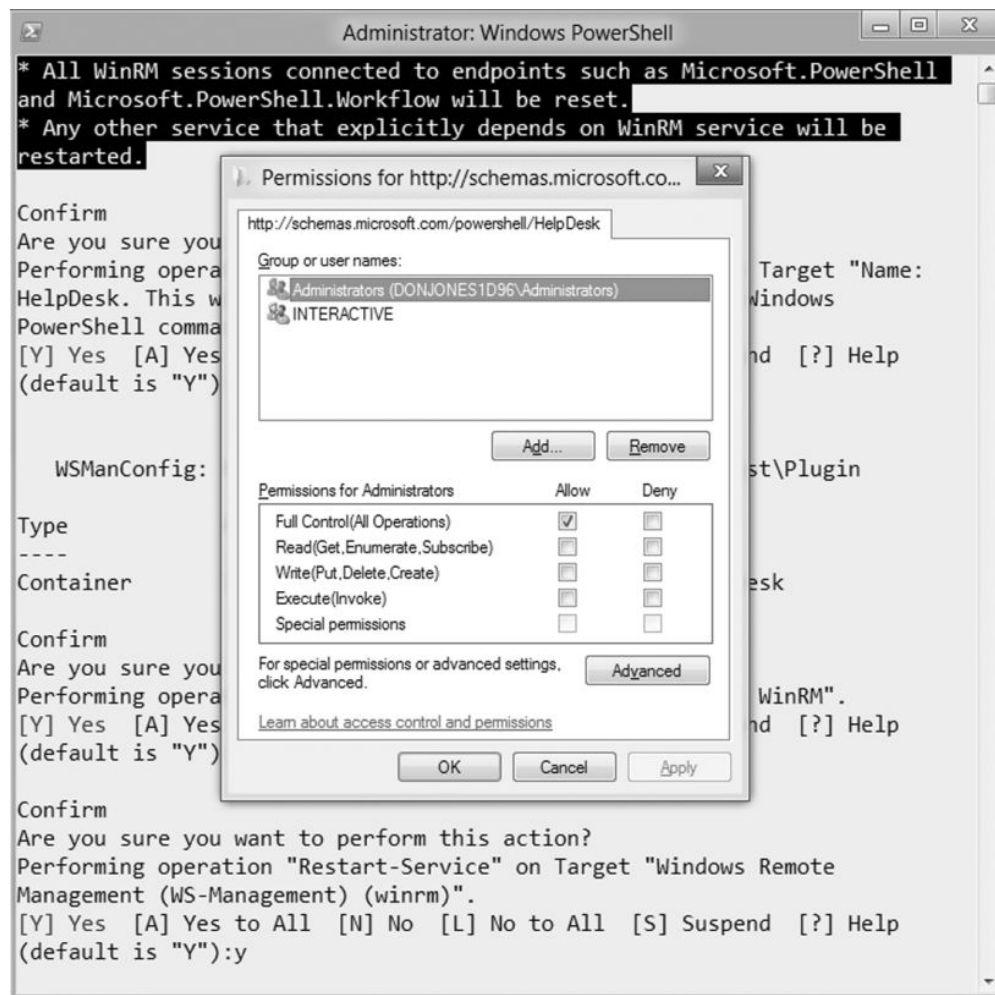


图23.2 设置端点权限

基于我们完成的内容，可以看到下述输出（截断后的），使用新端点的用户只能使用非常有限的命令。

```
PS C:\> Enter-PSSession -ComputerName DONJONES1D96 -
ConfigurationName HelpD
esk
[DONJONES1D96]: PS>Get-Command

Capability    Name
ModuleN
ame
-----
-----
CIM           Disable-NetAdapter
```

NetA...	
CIM	Disable-NetAdapterBinding
NetA...	
CIM	Disable-NetAdapterChecksumOffload
NetA...	
CIM	Disable-NetAdapterEncapsulatedPacketTaskOffload
NetA...	
CIM	Disable-NetAdapterIPsecOffload
NetA...	
CIM	Disable-NetAdapterLso
NetA...	
CIM	Disable-NetAdapterPowerManagement
NetA...	
CIM	Disable-NetAdapterQos
NetA...	
CIM	Disable-NetAdapterRdma
NetA...	
CIM	Disable-NetAdapterRsc
NetA...	
CIM	Disable-NetAdapterRss
NetA...	
CIM	Disable-NetAdapterSriov
NetA...	
CIM	Disable-NetAdapterVmq
NetA...	
CIM	Enable-NetAdapter
NetA...	
CIM	Enable-NetAdapterBinding
NetA...	
CIM	Enable-NetAdapterChecksumOffload
NetA...	
CIM	Enable-NetAdapterEncapsulatedPacketTaskOffload
NetA...	
CIM	Enable-NetAdapterIPsecOffload
NetA...	
CIM	Enable-NetAdapterLso
NetA...	
CIM	Enable-NetAdapterPowerManagement
NetA...	
CIM	Enable-NetAdapterQos
NetA...	

通过这种方式限制某个用户组能够使用的功能非常好。正如我们做的测试那样，他们甚至不必从控制台会话连接到**PowerShell**，他们可以使用基于**PowerShell**远程的GUI工具。这类工具的底层是使用的上述命令，利用这种技术给予用户使用某些功能的权限再好不过。

23.3 启用多跳远程 (multi-hop remoting)

该主题已经在第13章中简单提到，但该主题值得进一步深入。图23.3描述了“第二跳”或“多跳”的问题：从计算机A开始，并创建了一个PowerShell会话连接到计算机B。这是第一跳，通常该步骤可以正常工作。但当请求由计算机B再次创建第二跳，或是说连接到计算机C时，操作失败。

问题是由于PowerShell将凭据由计算机A委托到计算机B时出现的。所谓委托，是使得计算机B以你的身份运行任务的过程，因此确保你可以在计算机B上做任何有权限做的事，但不能做权限之外的事。默认情况下，委托只能传输一跳；计算机并没有权限将你的凭据委托给第三台计算机，也就是计算机C。

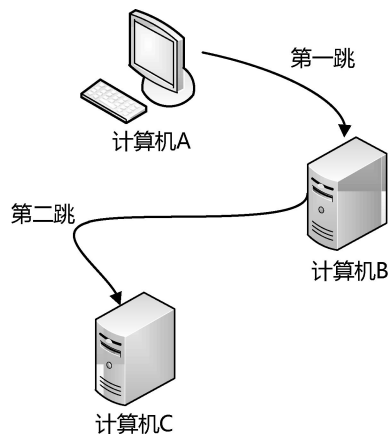


图23.3 在Windows PowerShell中的多跳远程

在Windows Vista以及之后版本，你可以启用多跳委托。该过程需要两步：

- (1) 在你的计算机（比如计算机A）上，运行`Enable-WSManCredSSP-RoleClient-DelegateComputer x`。可以将x替换为希望将身份委托到的计算机名称。你可以指定具体的计算机名称，当然也可以使用通配符。我们不推荐使用*，这会导致一些安全问题，但是可以对整个域进行授权，比如`*.company.com`。

- (2) 在第一跳连接到的计算机（比如计算机B）上，运行`Enable-WSManCredSSP-RoleServer`。

通过上述命令所做的变更，将会应用到计算机的本地安全策略；你也可以通过组策略手动进行变更，在较大的域环境中可能需要这么做。通过组策略管理这些超过了本章篇幅，但你可以通过**Enable-WSManCredSSP**的帮助信息获得更多信息。Don还写过一本“**Secrets of PowerShell Remoting guide**”，在该书中对策略相关的元素进行了更详细的阐述。

23.4 深入远程身份验证

我们发现，很多人都会认为身份验证是一个单向的过程：当你访问远程计算机时，你必须在登录该计算机之前提供你的凭据。但PowerShell远程采用了双向身份验证，这意味着远程计算机必须向你证明它的身份。换句话说，当你执行**Enter-PSSession -computerName DC01**时，名称为DC01的计算机必须在连接建立完成之前证明它就是DC01。

为什么？正常情况下，你的计算机将会通过域名系统（**Domain Name System, DNS**）将计算机名称（比如说DC01）解析为IP地址。但DNS可能会受到电子欺骗的攻击，因此不难想象，攻击者会攻入并将DC01的入口指向另一个IP地址——一个受攻击者控制的IP地址。你可能在不知情的情况下连接到DC01，实际上是一台冒名顶替的计算机，然后将你的凭据委托给这台冒名顶替的计算机——该倒霉了！双向身份验证会防止这类事发生：如果你连接到的计算机无法证明它就是那台你希望连接到的计算机，远程连接将会失败，这是好事——因此你不会希望在没有周密计划和考虑的情况下将这种保护关掉。

23.4.1 双向身份验证默认设置

微软期望对于PowerShell的使用大多是在域环境下。因此可以通过活动目录列出的实际计算机名称连接到计算机，域会为你处理双向身份验证。由域处理双向身份验证还会发生在访问其他可信任的计算机时。该技巧需要你为PowerShell提供的计算机名称满足以下两点要求。

- 名称可以被解析为IP地址。
- 名称必须与活动目录中的计算机名称匹配。

提供你所在的域的计算机名称，而对于可信域需要提供完全限定名（也就是计算机和域名称，比如DC01.COMPANY.LOC），这样远程通常就会生效。但你如果提供的是IP地址，或者需要提供与DNS中不同的名称（比如说CNAME别名），那么默认的双向身份验证将无法正常工作。因此你只有如下两种选择：SSL或是“受信任的主机”。

23.4.2 通过SSL实现双向身份验证

使用SSL，你必须获得目标计算机的SSL数字证书。证书颁发给的计算机名称必须与你输入访问的计算机名称相同。也就是说，如果你运行Enter-PSSession -computerName DC01.COMPANY.LOC -UseSSL -credential COMPANY\Administrator，那么安装在DC01上的证书必须颁发给“dc01.company.loc”，否则整个过程就会失败。注意，-credential参数在该场景中是强制参数。

在获取到证书之后，还需要将其安装到当前用户下的个人证书存储目录——通过微软管理控制台（Microsoft Management Console, MMC）界面是导入证书的最佳方式。仅仅是双击证书，通常情况下也能够将证书导入到账户的个人目录之下，但不通过MMC导入证书对SSL连接不会生效。

在完成证书安装之后，你需要在计算机上创建一个HTTP侦听器，并告诉侦听器使用刚刚安装的证书。而详细的指导教程会很长。由于这并不是大部分人会去配置的工作，我们在此不会将这部分内容包含在内。查看Don的Secrets of PowerShell Remoting guide（免费），你可以在此书中找到包含截图的详细教程。

23.4.3 通过受信任的主机实现双向身份验证

该技术比使用SSL证书略微简单，需要的配置步骤也会少很多。但该方式更加危险，这是由于该技术主要是对于选定的主机关闭双向身份验证。在开始之前，你需要能够自信地声明“不会有任何人会冒充这几台主机中的任何一台，或者入侵DNS记录”。对于在内部局域网的计算机来说，你也许会非常有自信这么声明。

然后你需要在没有双向身份验证的情况下识别计算机的另一种方式。在一个域中，这或许是类似“*.COMPANY.COM”这样在

Company.com域中的所有主机。

这是你需要配置整个域设置的一个实例，所以我们给你一个操作组策略的指南。该指南对于单机中的本地安全策略同样有效。

在任意GPO或本地计算机策略编辑器中，执行这些步骤：

- (1) 展开计算机配置。
- (2) 展开管理模板。
- (3) 展开Windows组件。
- (4) 展开Windows远程管理。
- (5) 展开WinRM客户端。
- (6) 双机受信任的主机。

(7) 启用策略并添加信任的主机列表，多个条目可以通过逗号分隔，比如“*.company.com,*.sales.company.com。”。

注意：旧的Windows版本可能没有在本机计算机策略中显示这些设置所需的模板，旧的域控制器的组策略对象中或许没有这些设置。对于这种情况，你可以在PowerShell中修改受信任的主机。在Shell中运行`help about_remote_troubleshooting`获取帮助。

现在你就可以在没有双向身份验证拦截的情况下连接到这些计算机。所有用于连接到这些计算机的远程命令中必须提供-Credential参数——如果不这么做，可能会导致连接失败。

23.5 动手实验

注意：

的计算机。

对于本次动手实验来说，你需要运行PowerShell v3或更新版本PowerShell

在本地计算机创建一个名称为**TestPoint**的端点。将端点配置为只有**SmbShare**组件会被自动载入，但该组件只有**Get-SmbShare**命令可见。同时要确保类似**Exit-PSSession**的关键**Cmdlet**可见，但不允许使用其他核心**PowerShell Cmdlet**。

通过**Enter-PSSession**（指定**localhost**作为计算机名称，**TestPoint**作为配置名称）连接到该端口，对该端口进行测试。当连接成功后，运行**Get-Command**，从而确保只有少数配置可见的命令可以被发现。

注意： 本次动手实验可能只在**Windows 8**、**Windows Server 2012**以及更新版本的**Windows**上可做——**SmbShare**组件并没有随更旧版本的**Windows**一起发行。

第24章 使用正则表达式解析文本

正则表达式是令人尴尬的主题之一。经常有学生让我们解释该概念——在解释的过程中才发现他们完全不需要正则表达式。正则表达式（**regular expression**，或**regex**）在文本解析的过程中非常有用，你经常会在**Unix**或**Linux**操作系统中用到。在**Power Shell**中，你会倾向于尽量少用文本解析——我们也发现你很少需要用到正则表达式。也就是说，我们当然知道某些时候在**PowerShell**中，你需要解析一些类似**IIS**日志的文本内容。这也是我们在本文阐述正则表达式的使用方式——用于解析文本文件。

别理解错我们的意思；你可以用正则表达式做更多的事情，我们将在本章结束之前阐述其中一部分。为了确保你有一个正确的期望，我们事先声明，我们在本书中将不会从宽度和深度方面尝试覆盖正则表达式的方方面面。正则表达式可以非常复杂。其自身就是一个完整的技术体系。我们将会把知识通过以直接应用到实践的方式传授给你，从而帮助你起步。在此之后，我们会给你一个方向，使你可以进一步自学。这些就足够了。

本章的目标是以最简单的方式将正则表达式的语法介绍给你，并且展示**PowerShell**是如何使用正则表达式的，如果你希望探索更加复杂的表达式，当然更好。这里我们将教会你如何在**Shell**中使用正则表达式。

24.1 正则表达式的目标

正则表达式需要以非常细节的语言书写，其目标是为了定义文本模型。比如说，**IPv4**地址以1~3位的数字为一组，一共4组。通过正则表达式可以定义该模式。虽然定义后还是会有**211.193.299.299**这样的非法地址，但这应该归于识别文本模式与数据有效范围的区别。

正则表达式最大的使用场景之一，也就是我们本章涵盖的内容——在一个类似日志这样大的文本文件中检测特定的文本模式。举例来说，你希望通过正则表达式在一个Web服务器日志文件中找到代表HTTP 500的特定文本，或是在一个SMTP服务器日志文件中寻找电子邮件地址。除了检测文本模式之外，还可以使用正则表达式捕捉匹配的文本，让你从日志文件中取出邮件地址。

24.2 正则表达式入门

最简单的正则表达式就是你希望匹配的文本字符串。比如“Don”，从技术角度来说，这就是一个正则表达式，在PowerShell中匹配“DON”“don”“Don”“DoN”等——PowerShell默认的匹配规则是不区分大小写。

某些特定的字符在正则表达式有特殊的含义，这些特定字符可以允许你检测文本变量中的文本模式。下面是一些示例。

- `\w` 用于匹配“文本字符”，也就是字母、数字以及下划线，但不包含标点符号和空格。正则表达式`\won`可以匹配“Don”“Ron”以及“ton”，`\w`可以代表任意字母、数字或下划线。
- `\W` 与 `\w` 相反（这也是PowerShell会区分大小写的一个示例），意思是它将会匹配空格与标点符号——按照“非字母”。
- `\d` 用于匹配包括0到9的任意数字。
- `\D` 用于匹配任意非数字。
- `\s` 用于匹配任意空格字符，比如Tab、空格或者回车符。
- `\S` 用于匹配任意非空格字符。
- `.`（句号）代表任意单个字符。
- `[abcde]`用于匹配在该集合中的任意字符。正则表达式`d[aeiou]n`可以匹配“Don”“Dan”，但不会匹配“Doun”或“Deen”。
- `[a-z]`匹配在此范围内的一个或多个字符，你可以使用逗号分隔列表指定多个范围，比如说`[a-f,m-z]`。
- `[^abcde]` 用于匹配不在该集合中的一个或多个字符，意味着正则表达式`d[^aeiou]`可以与“dns”匹配，但无法与“don”匹配。
- 将`?`置于另一个字母或特殊符号之后，可以用于匹配该字符的一个实例。所以正则表达式`do?n`可以与“don”匹配，但不会与“doon”匹配。该正则表达式还可以与“dn”匹配，这是由于`?`还可以代表空实例。

- * 用于匹配该符号之前任意数量的实例。正则表达式`do*n`将会与“doon”和“don”匹配。该正则表达式还可以与“dn”匹配，这是由于*还可以代表空实例。
- + 用于匹配该符号之前任意数量的实例。你会经常见到该字符和括号一起使用，从而创建了一种子表达式。举例来说，正则表达式`(dn)+o`可以与“dndndndno”匹配，这是由于该正则表达式可以重复匹配子表达式“dn”。
- \ (反斜杠) 是正则表达式转义字符。将该字符置于在正则表达式中有特殊意义的字符之前，从而使得该字符变为该字符的字面意思。比如，正则表达式`.`仅仅匹配一个句号，而不是像正常情况那样用于代表任意单个字符。如果希望匹配反斜杠，那么在反斜杠之前再加一个反斜杠：`\\`。
- {2} 用于匹配该符号之前特定数量的实例。比如，`\\d{1}`用于匹配1个数字。使用`{2, }`匹配2或多个数字，使用`{1, 3}`匹配至少1个但不超过3个实例。
- ^ 用于匹配字符串开始部分。比如，正则表达式`d.n`既可以匹配“don”，又可以匹配“pteranodon”。而正则表达式`^d.n`只能匹配“don”，而无法匹配“pteranodon”。这是由于^使得匹配只能发生在字符串开始，而^与[]共同使用时表达取匹配的反义。
- \$ 用于匹配字符串结尾部分。比如，正则表达式`.icks`既可以与“hicks”匹配，又可以与“sticks”（本例该匹配其实匹配的是“ticks”）匹配，还能够与“Dickson”匹配。但正则表达式`.icks$`无法与“Dickson”匹配，这是因为\$表示字符“s”应该是该字符串的最后一个字符。

总之，你快速查看了一遍正则表达式的语法。正如我们在开始所写的那样，正则表达式还有大量内容，但这些内容足够你完成基本工作。让我们来看一些正则表达式的例子：

- `\\d{1,3}.\\d{1,3}.\\d{1,3}.\\d{1,3}` 可以匹配IPv4地址的模式，但该表达式可以接受“432.567.875.000”这样的非法地址，也可以接受“192.169.15.12”这样的合法地址。
- `\\\\w+(\\\\w+)+` 可以匹配通用命名惯例（UNC）路径。大量的反斜杠使得该正则表达式难以阅读，这也是为什么在将正则表达式部署到生产环境之前对正则表达式进行调试和调整。
- `\\w{1}.\\w+@company.com` 可以匹配特定类型的电子邮件地址：首先是一个字母，然后是句号，最后是“@company.com”。比如

d.jones@company.com可以与该正则表达式进行匹配，“donald.jones@company.com.org”也能够匹配。我们将正则表达式能够匹配的部分进行加粗——正则表达式允许在匹配文本的开始或结尾存在额外的字符。在这种情况下就可以考虑使用^或\$。

注意：你可以通过在PowerShell运行help about_regular_expressions，发现更多关于正则表达式的基本语法。在本章末尾，我们将为你更进一步学习提供一些额外的资源。

24.3 通过-Match使用正则表达式

PowerShell包含一个比较运算符-Match，以及一个区分大小写的版本-Cmatch。通过这两个运算符与正则表达式进行比较。下面是一些示例。

```
PS C:\> "don" -match "d[aeiou]n"
True
PS C:\> "dooon" -match "d[aeiou]n"
False
PS C:\> "dooon" -match "d[aeiou]+n"
True
PS C:\> "djinn" -match "d[aeiou]+n"
False
PS C:\> "dean" -match "d[aeiou]n"
False
```

虽然使用正则表达式的方法很多，但我们主要依靠-Match测试正则表达式并确保正则表达式能够正确生效。如你所见，左边是你希望测试的字符串，右边是正则表达式。如果两边匹配，那么输出True；如果两边不匹配，那么输出False。

动手实验：是时候停止阅读并尝试使用-Match运算符了。运行一些之前我们在语法小节给你的示例，并确保你能够在Shell中将-Match运算符运用得得心应手。

24.4 通过Select-String使用正则表达式

现在我们终于到了本章的精华之处。我们使用一些IIS日志文件作为示例，这是由于IIS日志是适合正则表达式处理的纯粹的文本文件。如果能将这些日志以面向对象的形式读取到PowerShell中，那再好不过。可惜不能.....所以只能使用正则表达式。

动手实验： 如果你希望跟随练习，我们已经将示例日志文件压缩为zip，并在<http://MoreLunches.com> 网站上提供下载。只需查看本书的Web页面，你就可以发现示例的IISlog文件。本章中，我们将日志文件存入C:\Logfiles；该目录包含三个子目录（分别为WSSVC1、WSSVC2、WSSVC3），每个子目录包含一个日志文件。

让我们在日志文件中查找40x错误作为开头。这类错误主要是“找不到文件”以及其他错误，我们希望为Web开发人员生成一个缺失文件的报表。日志文件中，每一个HTTP请求为一行，每行又被分为以空格分割的域。我们还有一些文件包含“401”等作为其文件名的一部分，比如“error401.html”，我们不希望这部分结果出现在我们的结果中。我们将会指定一个类似\s40[0-9]\s的正则表达式，因为通过在40x错误之前和之后匹配空格，该表达式将能够匹配从400到499的错误。下面是我们使用的命令。

```
PS C:\logfiles> get-childitem -filter *.log -recurse | select-  
string -pattern  
    "\s40[0-9]\s" | format-table Filename,LineNumber,Line -wrap
```

注意，我们将当前目录变更为C:\logfiles，从而可以运行命令。我们通过寻找所有以.log结尾的文件，并递归查找子目录。这可以确保我们所有的日志文件都可以被包含在输出结果之内。接下来我们使用Select-String，并提供正则表达式作为参数。该命令的结果将会是一个类型为MatchInfo的对象；我们使用Format-Table命令，使得显示结果包含文件名称、行号以及包含匹配结果的文本。这使得找到缺失文件非常容易。然后我们将报表给予Web开发人员。

接下来，我们希望扫描所有被基于Gecko浏览器访问过的文件。开发人员告诉我们，使用该浏览器访问我们的网站的用户会遇到一些问题，他们希望找到具体被访问的文件。他们还将问题范围缩

减为使用Windows NT6.2操作系统运行浏览器的用户，这意味着我们需要在user-agent中寻找类似下面的字符串。

```
(Windows+NT+6.2;+WOW64;+rv:11.0)+Gecko
```

开发人员强调是否为64位操作系统无关紧要，因此我们不希望User-agent中仅是包含“WOW64”的结果。最终我们得到这个正则表达式：6.2;[\w\W]++Gecko——让我们对其进行分解。

- 6\.; ——这就是“6.2”；我们使用转义字符将句号变为字面意思上的句号，而不是作为单字符的通配符。
- [\w\W]+ ——一个或多个字符或非字符——换句话说是什么内容。
- +Gecko ——也就是字面意义上的加号，然后是“Gecko”。

下面是从日志文件返回匹配行的命令，还包含前几行的返回结果。

```
PS C:\logfiles> get-childitem -filter *.log -recurse | select-  
string -pattern  
    "6\.;[\w\W]++Gecko"  
  
W3SVC1\u_ex120420.log:14:2012-04-20 21:45:04 10.211.55.30 GET  
/MyApp1/  
    Testpage.asp - 80 - 10.211.55.29 Mozilla/  
    5.0+  
(Windows+NT+6.2;+WOW64;+rv:11.0)+Gecko/20100101+Firefox/11.0 200 0  
0  
    1125  
W3SVC1\u_ex120420.log:15:2012-04-20 21:45:04 10.211.55.30 GET  
/TestPage.asp-  
    80 - 10.211.55.29 Mozilla/5.0+  
(Windows+NT+6.2;+WOW64;+rv:11.0)+Gecko/  
    20100101+Firefox/11.0 200 0 0 1 109
```

这次我们保持输出结果为默认格式，而不是将结果发送给用于格式化的Cmdlet。

在最后一个例子中，我们将IIS日志文件变为Windows安全日志。事件日志实体中包含**Message**属性，该属性中包含关于事件的详细信息。不幸的是，该信息并没有良好格式化以便于人们阅读，也不易于计算机解析。我们希望查找所有事件ID为4624的事件，该事件代表账户登录事件（该ID代表的含义可能根据Windows版本的不同而有所不同；我们的示例是在Windows Server 2008 R2上）。但我们只希望查看以“WIN”开头的账户名称的登录信息，这些账户都是与在域中的计算机账户相关。另外，我们还要求账户结尾必须是从TM20\$到TM40\$，这些是我们感兴趣的特定计算机。我们需要的正则表达式大概如下：WIN[\W\w]+TM[234][0-9]\\$ ——注意我们需要使用转义符号将末尾的\$进行转义，因此该符号不会被解释成字符串结尾规则。我们需要包含[\W\w]（非字符和字符），这是由于我们的账户名称中可能包含连字符，该连字符无法与\w字符类匹配。因此最终下面是我们的命令。

```
PS C:\> get-eventlog -LogName security | where { $_.eventid -eq 4624 } |  
    select -ExpandProperty message | select-string -pattern  
    "WIN[\W\w]+TM[234][0-9]\$"
```

在开始部分，我们使用Where-Object，从而使得仅ID为4624的事件被筛选出来。然后我们将Message属性的内容存入纯字符串，并通过管道将其传输给Select-String。注意，这将会输出匹配的信息文本；如果我们的目标是输出所有匹配的事件，我们需要使用另一种方式。

```
PS C:\> get-eventlog -LogName security | where { $_.eventid -eq 4624 -and  
    $_.message -match "WIN[\W\w]+TM[234][0-9]\$" }
```

这里，我们不是输出Message属性的内容，而是查找Message属性匹配正则表达式的记录——接下来输出整个Event对象。接下来的命令就取决于结果希望输出的形式了。

24.5 动手实验

注意:

对于本次动手实验来说，你需要运行PowerShell v3或更新版本PowerShell的计算机。

请不要会错意，正则表达式的复杂程度可以让你头痛，所以请不要开始就尝试创建复杂的正则表达式——从简单开始。下面一些练习可以帮助你入门。使用正则表达式和运算符完成下列任务。

- 获取活动目录中所有名称包含2位数字的文件。
- 获得计算机中所有非微软的进程，并显示进程ID、名称以及公司名称。提示：通过管道将Get-Process传递给Get-Member，从而显示属性名称。
- 在Windows Update日志中，该日志通常位于C:\Windows，你只希望显示代理开始安装文件的日志行。你或许需要在记事本中打开日志文件，从而找出你需要选择的字符串。

24.6 进一步学习

你将会在PowerShell的其他地方发现使用正则表达式，其中很多地方包含本书未提到的Shell元素。下面是一些示例。

- Switch脚本构造器中包含一个参数，使得其值可以与一个或多个正则表达式进行比较。
- 高级脚本和函数（脚本Cmdlets）可以使用一个基于正则表达式的输入验证工具防止无效的参数值。
- -Match运算符（在本章简单介绍）将字符串与正则表达式进行对比。还有一部分未做介绍——将匹配的字符串捕捉存入一个自动的\$matches集合。

PowerShell使用业界标准的正则表达式。如果你希望更深入地学习，我们推荐你阅读由Jeffrey E.F. Friedl 著的*Mastering Regular Expressions* by Jeffrey (O'Reilly 出版社)。市场上还有大量的正则表达式书籍，其中一部分只面向Windows和.NET（也就面向PowerShell），其中一部分书籍专注针对具体场景构建正则表达式，等等。请浏览你喜欢的在线书店，从而查找是否存在吸引你或满足你特定需求的书籍。

我们也使用免费的在线正则表达式资源：<http://RegExLib.com>。该网站包含用于不同目的的大量正则表达式示例（电话号码、邮件地

址、IP地址等)。我们还使用<http://RegExTester.com> 这个网站测试我们的正则表达式，从而确保正则表达式能够满足我们的需求。

第25章 额外的提示、技巧以及技术

到目前为止，你已经快结束为期一个月的学习了。下一章是对你最后的一个测验。在该测试中，你需要从头开始完成一个完整的管理任务。但是在进行这项任务之前，我们想给你分享一些额外的提示以及技巧来完成这次学习之旅。

25.1 Profile、提示以及颜色：自定义Shell界面

每一个PowerShell进程开启时都是一样的：一样的别名，一样的PSDrives，一样的色彩等。为什么不使用自定义的Shell界面呢？

25.1.1 PowerShell Profile脚本

在前文中，我们阐述了PowerShell主机应用程序和PowerShell引擎本身的区别。PowerShell的主机应用程序，比如PowerShell ISE的控制台，是指将命令发送至PowerShell引擎的一种方式。首先PowerShell引擎会执行命令，然后主机应用程序再显示执行的结果。主机应用程序的另一个功能是当新开一个Shell窗口时，载入和运行Profile脚本。

这些Profile脚本可被用作自定义PowerShell的运行环境——载入SnapIn管理单元或者模块，切换到另外的根路径，定义需要使用的功能等。例如，下面是Don在计算机上使用的一个Profile脚本。

```
Import-Module ActiveDirectory
Add-PSSnapIn SqlServerCmdletSnapIn100
Cd C:\
```

该Profile载入了Don最常用的两个Shell的扩展程序，并且修改根路径为C盘——C盘也是Don喜欢使用的根路径。当然，你可以将你喜欢的任意命令放入Profile脚本中。

注意： 你可能认为没有必要载入ActiveDirectory模块，因为当用户尝试使用包含在该模块中的任一命令时，该模块会被隐式载入。该模块也会映射到一个AD:PSDrive，Don希望当新开一个Shell窗口时，该AD:PSDrive就处于可用状态。

在PowerShell中，并没有默认的Profile脚本存在，你创建的Profile脚本会依赖于你期望该脚本的工作方式。如果你需要查看详细信息，那么请执行Help About_Profiles。当然，你最需要考虑的是，是否会用到多种PowerShell的主机应用程序。比如，我们倾向于在常规控制台和PowerShell ISE中来回切换，我们希望这两种主机应用程序都会运行相同的Profile脚本，所以需要确保在正确的路径下创建正确的Profile脚本。同时，我们也必须验证Profile脚本中的命令，因为该Profile脚本都会应用到控制台以及ISE主机应用程序——比如一些调整色彩等控制台设置的命令在ISE中可能会运行失败。

下面是控制台主机尝试载入的一些文件，以及尝试载入这些文件的顺序。

(1) \$PsHome/Profile.PS1——不管使用何种主机应用程序，该脚本都会对计算机上所有用户执行（请记住，\$PSHome路径是已经被预定义在PowerShell中，并且包含PowerShell的安装文件夹的路径）。

(2) \$PsHome/Microsoft.PowerShell_Profile.PS1——如果计算机上的用户均使用控制台主机应用程序，那么该脚本会对所有用户执行。如果他们使用的是PowerShell的ISE，那么\$PsHome/Microsoft.PowerShellISE_Profile.ps1脚本会被执行。

(3) \$Home/Documents/WindowsPowerShell/Profile.PS1——该脚本仅会对当前用户执行（因为该脚本存在于用户的根目录下），不管该用户使用的是何种主机应用程序。

(4) \$Home/Documents/WindowsPowerShell/Microsoft.PowerShell_Profile.ps1——仅会针对当前使用PowerShell控制台的用户使用。如果用户使用的是PowerShell ISE，那么会执行\$Home/Documents/WindowsPowerShell/Microsoft.PowerShellISE_Profile.PS1。

如果上面脚本中某一个或者几个不存在，那么也没关系。主机应用程序会跳过不存在的脚本，继续寻找下一个可用的脚本。

在64位操作系统上，由于存在独立的32位与64位的PowerShell程序，所以脚本也会包含32位与64位的版本。请不要期望相同的脚本在32位与64位PowerShell中都能正常运行。这意味着，某些模块或者扩展程序仅在某一个架构中才可用，所以请不要尝试使用一个32位的Profile脚本将某个64位的模块载入32位的PowerShell中，因为这根本不可能成功。

请注意，`About_Profiles`的帮助文档与我们上面罗列的有一点不同。但是我们的经验可以证明，上面的列表是正确的。下面是针对该列表的其他一些知识点。

- `$PsHome`是包含PowerShell安装路径信息的内置变量；在大部分操作系统中，该变量的值是
`C:\Windows\System32\WindowsPowerShell\V1.0`（针对64位操作系统上64位版本的PowerShell）。
- `$Home`是另一个内置的变量，该变量指向当前用户的配置文件夹（比如`C:\Users\Administrator`）。
- 在前面的列表中，我们使用“Documents”来表示文档文件夹，但是在某些版本Windows系统中可能是“My Documents”。
- 在前面的列表中写到“不管用户使用何种主机应用程序”，从技术上讲并不恰当。准确地说，针对微软发布的主机应用程序（控制台或者ISE），该命题正确；但是针对非微软发布的主机应用程序，根本无法使用该规则。

因为期望将相同的Shell扩展程序载入到PowerShell，而不管使用控制台还是ISE，所以我们选择自定义

`$Home\Documents\WindowsPowerShell\Profile1.PS1`——因为该Profile脚本在微软提供的两种主机应用程序中都可以运行。

动手实验：为什么你自己不尝试创建一个或者多个Profile脚本呢？即使在这些脚本中仅打印出一些简单的信息，比如“`It Worked`”，这是查看不同脚本执行的一个好方法。但是请记住，你必须选择使用Shell（或者ISE），并且需要重新打开该Shell（或者ISE）去检查Profile脚本是否运行。

请记住，Profile脚本也仅是脚本而已，它会依赖于PowerShell的当前执行策略。如果设置的执行策略是Restricted，那么Profile脚本就不能运行；如果设置的执行策略是AllSigned，那么你的Profile脚本必须经过

签名才能运行。在第17章中讲到了执行策略以及脚本签名部分。如果你忘记了该知识点，请回到第17章重新学习。

25.1.2 自定义提示

PowerShell提示——也就是你在本书中看到的PS C:\>这类字符，是由一个名为提示（**Prompt**）的内置函数产生的。如果你希望自定义该提示，很简单，只需要替换该函数即可。可以在**Profile**脚本中定义一个新的提示函数，这样在你每次打开**Shell**界面的时候都可以采用新的提示函数。

下面是默认的提示函数。

```
Function Prompt
{
    $(IF (Test-Path Variable:/PSDebugContext) { '[DBG]: ' }
    ELSE { ' ' }) + 'PS ' + $(Get-Location) `
    + $(IF ($NestedPromptLevel -Ge 1) { '>>' }) + '> '
}
```

该函数首先会检测\$DebugContext变量是否被预定义在PowerShell的Variable:Drive中。如果有，那么该函数就会将[DBG]:添加到提示启动阶段。否则，该提示会被定义为PS再加上由Get-Location Cmdlet返回的当前路径（比如PS D:\Test>）。如果该Shell处于嵌套提示中——由内置函数\$NestedPromptLevel返回，那么提示中会添加“>>”字样。

下面是自定义的一个提示函数。你可以直接将该函数加入到任意Profile脚本中，这样可以保证后续新开启的Shell进程都会将该提示作为一个标准提示函数使用。

```
Function Prompt {
    $Time = (Get-Date).ToShortTimeString()
    "$Time [$ENV:COMPUTERNAME]:> "
}
```

该自定义函数会返回当前时间，后面接着当前计算机名称（计算机名称包含在方括号内）。

```
6:07 PM [CLIENT01]:>
```

在这里，通过双引号改变了PowerShell特定的行为——PowerShell会使用双引号中的内容来替换变量（比如\$Time）的值。

25.1.3 调整颜色

在前面的章节中，我们看到，当Shell界面报出很多错误时，我们觉得多么刺眼。当Don还是一个小孩的时候，他在英语课堂上总是很痛苦——因为他总是能看到汉森女士批改之后的文章（使用红笔标出的红色文字的提醒）。但是幸运的是，在PowerShell中，你可以修改使用的默认颜色选项。

默认文本前景色与后景色都可以通过单击PowerShell命令窗口左上角的边框来修改。选择“属性”，之后切换到“颜色”标签页，如图25.1所示。



图25.1 配置默认Shell界面颜色

修改错误、警告以及其他信息的颜色则更加复杂，需要通过运行命令才能实现。但是你可以将这部分命令放到**Profile**脚本中，这样每次进入**PowerShell**时，都会执行这些命令。比如下面的命令可以将错误消息的前景色修改为绿色，这样你可以觉得稍微舒缓一点。

```
(Get-Host).PrivateData.ErrorForegroundColor="Green"
```

我们可以通过命令修改下列设置的颜色。

- ErrorForegroundColor
- ErrorBackgroundColor
- WarningForegroundColor
- WarningBackgroundColor
- DebugForegroundColor
- DebugBackgroundColor
- VerboseForegroundColor
- VerboseBackgroundColor
- ProgressForegroundColor
- ProgressBackgroundColor

下面是你可以选择的几种颜色。

- Red
- Yellow
- Black
- White
- Green
- Cyan
- Magenta
- Blue

同时，也存在这些颜色的对应深色颜色：DarkRed，DarkYellow，DarkGreen，DarkCyan，DarkBlue等。

25.2 运算符: -AS,-IS,-Replace,-Join,-Split,-IN,-Contains

这些额外的运算符在多种情形下都非常有用，可以通过它们来处理数据类型、集合和字符串。

25.2.1 -AS和-IS

-AS运算符会将一种已存在的对象转换到新的对象类型，从而产生一个新的对象。例如，如果存在一个包含小数的数字（可能来自一个除法计算），你可以通过Converting或者Casting将这个数字转化为一个整数。

```
1000/3 -AS [INT]
```

语句的结构：首先是一个将被转换的对象，然后是-AS运算符，最后是一个中括号，中括号中包含转化之后的类型。这些类型可以是[String], [XML], [INT], [Single], [Double], [Datetime]等，罗列的这些类型应该不是你经常使用到的类型。从技术上讲，在该示例中，将数值转化为整数是指将小数部分通过四舍五入方式转为整数，而并不是简单地将小数部分去掉。

-IS运算符通过类似方式来实现。该运算符主要用来判断某个对象是否为特定类型，如果是，则返回True，否则为False。比如下面的这些示例：

```
123.45 -IS [INT]
"SERVER-R2" -IS [String]
$True -IS [Bool]
(Get-Date) -IS [DateTime]
```

动手实验： 请执行上面每一个命令，然后确认其返回结果。

25.2.2 -Replace

-Replace运算符主要用来在某个字符串中寻找特定字符（串），最后将该字符（串）替换为新的字符（串）。

```
PS C:\> "192.168.34.12" -Replace "34","15"  
192.168.15.12
```

命令的结构如下：首先是源字符串，之后为**-Replace**运算符。然后你要提供你需要在源字符串中寻找的字符（串），最后跟上一个逗号外加最新的字符（串）。在上面的示例中，我们将字符串中的“34”替换为“15”。

25.2.3 -Join和-Split

-Join和**-Split**运算符主要用作将数组转化为分隔列表和将分隔列表转化为数组。

例如，存在包含5个元素的数组：

```
PS C:\> $Array = "one","two","three","four","five"  
PS C:\> $Array  
one  
two  
three  
four  
five
```

因为PowerShell会自动将使用逗号隔开的列表识别一个数组，所以上面的命令可以执行成功。假如你现在需要将这个数组里的值转换为以管道符隔开的字符串，你可以通过**-Join**来实现。

```
PS C:\> $Array -Join "|"  
one|two|three|four|five
```

可以将该执行结果放入一个变量，这样可以直接重用，或者将其导出为一个文件。

```
PS C:\> $String = $Array -Join "|"
PS C:\> $String
one|two|three|four|five
PS C:\> $String | Out-File Data.DAT
```

同时，我们可以使用**-Split**运算符来实现相反的效果：它会从一个分隔的字符串中产生一个数组。例如，假如存在仅包含一行四列数据的一个文件，在该文件中以制表符对列进行隔离。将该文件的内容显示出来，类似下面这样。

```
PS C:\>Gc Computers.tdf
Server1 Windows East Managed
```

请记住，这里的**Gc**是**Get-Content**的别名。

你可以通过**-Split**运算符将该内容拆成4个独立的数组元素。

```
PS C:\> $Array = (Gc Computers.tdf) -Split "`t"
PS C:\> $Array
Server1
Windows
East
Managed
```

请注意，这里我们使用转义字符、一个重音符以及一个“t”(**`t**)来表示制表符。这些字符必须包含在一个双引号中，这样**PowerShell**才能识别该转义字符。

产生的数组中包含4个元素，你可以通过它的索引编号来单独查询对应元素。

```
PS C:\> $Array[0]
Server1
```

25.2.4 -Contains和-IN

-Contains运算符对PowerShell初学者而言可能会比较容易混淆。他们可能会尝试下面的脚本。

```
PS C:\> 'this' -Contains '*his*'
False
```

实际上，他们是期望运行-like运算符。

```
. PS C:\> 'this' -Like '*his*'
True
```

-Like运算符用来进行通配符比较运算。-Contains运算符主要用作在一个集合中是否存在特定对象。比如，创建包含多个字符串对象的一组集合，然后检查特定对象是否包含在该集合中。

```
PS C:\> $Collection = 'abc','def','ghi','jkl'
PS C:\> $Collection -Contains 'abc'
True
PS C:\> $Collection -Contains 'xyz'
False
```

-IN运算符会实现相同的功能，但是它会颠倒运算对象的顺序。也就是说，集合在右边，而需要检查的对象在左边。

```
PS C:\> $Collection = 'abc','def','ghi','jkl'
PS C:\> 'abc' -IN $Collection
True
PS C:\> 'xyz' -IN $Collection
False
```

25.3 字符串处理

假如存在一个字符串，你需要将该字符串全部转化为大写，或者你可能需要取得该字符串的最后三个字符。那么应该如何实现呢？

在PowerShell中，字符串是一种对象，所以就会存在多种方法（**Methods**）。一个方法是指对象可以完成的某项工作的方式，通常是针对对象本身。你可以通过将这个对象通过管道发送给Gm来查看该对象可用的方法。

```
PS C:\> "Hello" | Gm
```

TypeName: System.String		
Name	MemberType	Definition
Clone	Method	System.ObjectClone()
CompareTo	Method	int CompareTo(System.Object value...
Contains	Method	bool Contains(string value)
CopyTo	Method	System.VoidCopyTo(intsourceInde...
EndsWith	Method	bool EndsWith(string value), bool...
Equals	Method	bool Equals(System.Objectobj), b...
GetEnumerator	Method	System.CharEnumeratorGetEnumerat...
GetHashCode	Method	int GetHashCode()
GetType	Method	type GetType()
GetTypeCode	Method	System.TypeCodeGetTypeCode()
IndexOf	Method	int IndexOf(char value), intInde...
IndexOfAny	Method	int IndexOfAny(char[] anyOf), int...
Insert	Method	string Insert(intstart Index, str...
IsNormalized	Method	bool IsNormalized(), bool IsNorma...
LastIndexOf	Method	int LastIndexOf(char value), int ...
LastIndexOfAny	Method	int LastIndexOfAny(char[] anyOf),...
Normalize	Method	string Normalize(), string Normal...
PadLeft	Method	string PadLeft(int totalWidth), s...
PadRight	Method	string PadRight(int totalWidth), ...
Remove	Method	string Remove(int startIndex, int...
Replace	Method	string Replace(char oldChar, char...
Split	Method	string[] Split(Params char[] sepa...
StartsWith	Method	bool StartsWith(string value), bo...
Substring	Method	string Substring(int startIndex),...
ToCharArray	Method	char[] ToCharArray(), char[] ToCh...
ToLower	Method	string ToLower(), string ToLower(...
ToLowerInvariant	Method	string ToLowerInvariant()
ToString	Method	string ToString(), string ToStrin...
ToUpper	Method	string ToUpper(), string ToUpper(...
ToUpperInvariant	Method	string ToUpperInvariant()
Trim	Method	string Trim(Params char[] trimCha...
TrimEnd	Method	string TrimEnd(Params char[] trim...
TrimStart	Method	string TrimStart(Params char[] tr...
Chars	ParameterizedProperty	char Chars(int index) {get;}

Length	Property	System.Int32	Length {get;}
--------	----------	--------------	---------------

下面是一些比较有用的**String**方法。

- **IndexOf()**会返回特定字符在字符串中的位置。

```
PS C:\> "SERVER-R2".IndexOf("-")  
6
```

- **Split()**, **Join()**和**Replace()**类似于上面讲到的**-Split**, **-Join**和**-Replace**。但是我们更加倾向于使用**PowerShell**的运算符而不是**String**的方法。
- **ToLower()**和**ToUpper()**可以将字符串转化为小写或大写。

```
PS C:\> $ComputerName = "SERVER17"  
PS C:\> $ComputerName.ToLower()  
server17
```

- **Trim()**会将一个字符串前后的空格去掉；**TrimStart()**和**TrimEnd()**会将一个字符串的前面或者后面的空格去掉。

```
PS C:\> $UserName = "Don"  
PS C:\> $UserName.Trim()  
Don
```

上面这些方法都是处理或者修改**String**对象比较方便的方法。请记住，所有这些方法都可以运用在包含字符串的变量（比如前面的**ToLower()**和**Trim()**示例），也可以用在静态的字符串上（比如前面的**IndexOf()**示例）。

25.4 日期处理

和**String**类型对象一样，**Date**(如果你喜欢，也可以是**DateTime**)对象也可以使用多种方法进行处理。通过这些方法，我们可以对日期和时

间进行处理和计算。

```
PS C:\>Get-Date | Gm
```

```
    TypeName: System.DateTime
```

Name	MemberType	Definition
-----	-----	-----
Add	Method	System.DateTimeAdd(System.TimeSpan ...
AddDays	Method	System.DateTimeAddDays(double value)
AddHours	Method	System.DateTimeAddHours(double value)
AddMilliseconds	Method	System.DateTimeAddMilliseconds(doub...
AddMinutes	Method	System.DateTimeAddMinutes(double va...
AddMonths	Method	System.DateTimeAddMonths(int months)
AddSeconds	Method	System.DateTimeAddSeconds(double va...
AddTicks	Method	System.DateTimeAddTicks(long value)
AddYears	Method	System.DateTimeAddYears(int value)
CompareTo	Method	intCompareTo(System.Object value), ...
Equals	Method	boolEquals(System.Object value), bo...
GetDateTimeFormats	Method	string[] GetDateTimeFormats(), strin...
GetHashCode	Method	intGetHashCode()
GetType	Method	type GetType()
GetTypeCode	Method	System.TypeCodeGetTypeCode()
IsDaylightSavingTime	Method	bool IsDaylightSavingTime()
Subtract	Method	System.TimeSpanSubtract(System.Date...
ToBinary	Method	long ToBinary()
ToFileTime	Method	long ToFileTime()
ToFileTimeUtc	Method	long ToFileTimeUtc()
ToLocalTime	Method	System.DateTimeToLocalTime()
ToLongDateString	Method	string ToLongDateString()
ToLongTimeString	Method	string ToLongTimeString()
ToOADate	Method	double ToOADate()
ToShortDateString	Method	string ToShortDateString()
ToShortTimeString	Method	string ToShortTimeString()
ToString	Method	string ToString(), string ToString(s...
ToUniversalTime	Method	System.DateTimeToUniversalTime()
DisplayHint	NoteProperty	
Microsoft.PowerShell.Commands.Displa...		
Date	Property	System.DateTime Date {get;}
Day	Property	System.Int32 Day {get;}
DayOfWeek	Property	System.DayOfWeekDayOfWeek {get;}
DayOfYear	Property	System.Int32 DayOfYear {get;}
Hour	Property	System.Int32 Hour {get;}
Kind	Property	System.DateTimeKind Kind {get;}
Millisecond	Property	System.Int32 Millisecond {get;}
Minute	Property	System.Int32 Minute {get;}
Month	Property	System.Int32 Month {get;}

Second	Property	System.Int32	Second {get;}
Ticks	Property	System.Int64	Ticks {get;}
TimeOfDay	Property	System.TimeSpan	TimeOfDay {get;}
Year	Property	System.Int32	Year {get;}
DateTime	ScriptProperty	System.Object	DateTime {get=if ((&
			{...

请记住，通过上面列表中的属性可以访问一个**DateTime**的一部分数据，比如日期、年或者月。

```
PS C:\> (Get-Date).Month
10
```

上面列表中的方法可以实现两个功能：计算或者将**DateTime**转化为其他格式。例如，假如需要获取**90**天之前的日期，我们可以对**AddDays()**使用一个负数实现。

```
PS C:\> $Today=Get-Date
PS C:\> $90DaysAgo=$Today.AddDays(-90)
PS C:\> $90DaysAgo

2014年12月19日 9:36:47
```

名称中以“**To**”开头的方法可以实现将日期以及时间转化为某种特定格式，比如短日期类型。

```
PS C:\> $90DaysAgo.ToShortDateString()
2014/12/19
```

另外需要注意的是，这些方法都是依赖于你计算机本地的区域设定——区域设定决定日期和时间的特定格式。

25.5 处理WMI日期

在WMI中存储的日期和时间格式都难以直接利用。例如，Win32_OperatingSystem类主要用来记录计算机上一次启动的时间，其日期和时间格式如下。

```
PS C:\>Get-WMIObject Win32_OperatingSystem | Select LastBootUpTime
LastBootUpTime
-----
20150317090459.125599+480
```

PowerShell的开发人员知道直接使用这些信息会比较困难，所以他们对每一个WMI对象添加了一组转换方法。将WMI对象通过管道发送给Gm，请注意观察最后两个方法。

```
PS C:\>Get-WMIObject Win32_OperatingSystem | Gm
TypeName:
System.Management.ManagementObject#root\cimv2\Win32_OperatingSystem

Name           MemberType      Definition
-----
Reboot          Method          System.Management...
SetDateTime     Method          System.Management...
Shutdown        Method          System.Management...
Win32Shutdown   Method          System.Management...
Win32Shutdown Tracker Method          System.Management...
BootDevice      Property        System.String Boo...
...
PSStatus        PropertySet     PSStatus {Status,...
ConvertFromDateTime ScriptMethod    System.Object Con...
ConvertToDateTime ScriptMethod    System.Object Con...
```

将输出结果集中间的大部分信息去除，这样你能很轻易地发现后面的ConvertFrom DateTime()和ConvertToDateTime()方法。在该示例中，获取到的是WMI的日期和时间。假如需要转化为正常的日期和时间格式，请参照下面的命令。

```
PS C:\> $OS=Get-WMIObject Win32_OperatingSystem
PS C:\> $OS.ConvertToDateTime($OS.LastBootUpTime)
```


2015年3月17日 9:04:59

如果你期望将正常的日期和时间信息放入到一个正常表中，你可以通过**Select-Object**或者**Format-Table**命令来创建自定义计算列以及属性。

```
PS C:\> Get-WMIObject Win32_OperatingSystem |Select
BuildNumber,__Server,@{
l='LastBootTime';E={$_.ConvertToDateTime($_.LastBootUpTime)}}

BuildNumber          __Server          LastBootTime
-----
7601                SERVER-R2          2015/3/17
9:04:59
```

25.6 设置参数默认值

大多数PowerShell命令的一些参数都包含默认值。例如，运行**Dir**命令，默认会指向当前路径，而并不需要指定**-Path**参数。在第三版PowerShell之后（包含第三版），你可以对任意命令的任意参数——甚至是针对多个命令，指定自定义的默认值。当执行不带有指定参数的命令时，才会采用设定的默认值；但是当运行命令时有手动指定参数以及对应值，之前设定的默认值会被覆盖。

默认值保存在名为**\$PSDefaultParameterValues**的特殊内置变量中。当每次新开一个PowerShell窗口时，该变量均置空，之后使用一个哈希表来填充该变量（可以通过**Profile**脚本使得默认值始终有效）。

例如，假如你希望创建一个包含用户名以及密码的凭据对象，然后将该对象设置为所有命令中**-Credential**参数的默认值。

```
PS C:\> $Credential = Get-Credential -UserName Administrator -
Message
"Enter Admin Credential"
PS C:\> $PSDefaultParameterValues.Add('*:Credential',$Credential)
```

或者，如果仅希望Invoke-Command Cmdlet每次运行时都会提示需要凭据，此时请不要直接分配一个默认值，而是分配一段执行Get-Credential命令的脚本块。

```
PS C:\>$PSDefaultParameterValues.Add('Invoke-Command:Credential',
{Get-Credential -Message 'Enter Administrator Credential' -UserName
Administrator})
```

可以看到该Add()方法的基本格式：第一个参数为<Cmdlet>:<Parameter>，该<Cmdlet>可以接受*等通配符。Add()方法的第二个参数要么为直接给出的默认值，要么是执行其他（一个或多个）命令的脚本块。

你可以执行下面的命令，查看\$PSDefaultParameterValues包含的内容。

```
PS C:\>$PSDefaultParameterValues

Name                Value
-----
*:Credential        System.Management.Automation.PSCredential
Invoke-Command:Credential  Get-Credential -Message 'Enter
administ
```

补充说明

PowerShell的变量由作用域（Scope）控制。我们在第21章中简单介绍了作用域，同时作用域也会对参数默认值进行影响。

如果在命令行中设置了\$PSDefaultParameterValues，那么该参数会针对本Shell会话中的所有脚本以及命令起作用。但是如果仅在一段脚本中设置了>\$PSDefaultParameterValues，那么同样，也只会在该脚本作用域中起作用。该技术非常有用，因为这意味着你可以在一段脚本中设置多个参数的默认值，但是并不影响其他脚本或者Shell会话的运行。

作用域的核心思想是“无论脚本发生了什么，仅会影响该脚本”这个概念。如果你想深入研究作用域，请查阅About_Scope帮助文档中的详细内容。

你可以通过PowerShell中的About_Parameters_Default_Values帮助文档来查看该特性更多的知识点。

25.7 学习脚本块

脚本块是PowerShell的一个关键知识点。之前你可能已经能简单地使用脚本块了。

- Where-Object命令的-FilterScript参数会使用脚本块。
- ForEach-Object命令的-Process参数会使用脚本块。
- 使用Select-Object创建自定义属性的哈希表或者使用Format-Table创建自定义列的哈希表，都会需要一个脚本块作为E或者Expression的键值。
- 正如本章前面所讲，参数的默认值也可以为一个脚本块。
- 针对一些远程处理以及Job相关的命令，比如Invoke-Command和Start-Job命令，也需要一个脚本块作为-ScriptBlock参数的值。

那么，什么是脚本块呢？简单来讲，脚本块是指包含在大括号中的全部命令——哈希表除外（哈希表在大括号之前会带有@符号）。你可以在命令行中输入一个脚本块，然后将该脚本块赋值给一个变量，再使用&该调用运算符来执行该脚本块。

```
PS C:\> $Block = {
>> Get-Process | Sort -Property Vm -Descending | Select -First 10 }
>>
PS C:\> &$Block
```

Handles	NPM(K)	PM(K)	WS(K)	VM(M)	CPU(s)	Id	ProcessName
680	42	14772	13576	1387	3.84	404	svchost
454	26	68368	75116	626	1.28	1912	powerShell
396	37	179136	99252	623	8.45	2700	powerShell
497	29	15104	6048	615	0.41	2500	SearchIndexer
260	20	4088	8328	356	0.08	3044	taskhost
550	47	16716	13180	344	1.25	1128	svchost

1091	55	19712	35036	311	1.81	3056	explorer
454	31	56660	15216	182	45.94	1596	MsMpEng
163	17	62808	27132	162	0.94	2692	dwm
584	29	7752	8832	159	1.27	892	svchost

你可以使用脚本块来完成更多的工作。如果希望进一步学习脚本块，请参阅PowerShell中的About_Script_Block帮助文档。

25.8 更多的提示、技巧及技术

正如本章开始所说，本章只是展示一些需要让你知晓的知识点，但是这些知识点并未出现在之前的章节中。当然，在逐渐学习PowerShell的过程中，你会遇到更多的提示以及技巧，也会获得更多的经验。

你也可以订阅我们的Twitter: [too-@jeffhicks](#)和[@concentrateddon](#)。我们会定期在Twitter上分享一些有用的提示以及小技巧。PowerShell.Org网站上也提供邮件列表定期推送一些小技巧。有些时候，通过点滴的学习，你可以更容易在某技术领域成为专家，所以请将这些提示、技巧以及技术，包括以后会遇到的其他资源作为不断提高PowerShell水平的一种沉淀吧！

第26章 使用他人的脚本

尽管我们希望你能从头开始编写一些自己的PowerShell命令脚本，但是我们也意识到，在编写过程中你会严重依赖于互联网上的一些示例。不管你是直接利用别人博客中的示例还是修改在线脚本代码库——比如PowerShell代码库（<http://PoshCode.org>）中发现的脚本，其实能利用借鉴别人的PowerShell脚本也算作一项重要的核心技能。在本章中，我们会带领你学会通过该过程理解别人的脚本，并最终将脚本修改以适合我们的需要。

特别感谢： 感谢提供本章脚本的Christoph Tohermes和Kaia Taylor。我们特意让他们提供一些带有瑕疵的脚本，这些脚本与我们通常见到最佳实践中那些完美的脚本不一样。在某些情况下，我们甚至会故意将他们提供的脚本进行破坏，使得本章中的场景更真实。我们非常感激他们对该学习活动所做的贡献。

请注意，我们选择这些脚本主要是因为在这些脚本中，他们使用了一些在本书中并未涉及的高阶PowerShell功能。再次，我们需要说明，这就是真实的世界：你总是会碰到陌生的东西。本练习的一个目的是尽快知道某个脚本的功能，即便你并未学习过该脚本用到的所有技术。

26.1 脚本

代码清单26.1展示了名为New-WebProject.ps1的完整脚本。该脚本主要用于调用微软IIS Cmdlet——该Cmdlet存在于已安装Web服务角色的Windows Server 2008 R2以及之后版本的操作系统上。

代码清单26.1 New-WebObject.ps1

```
param(
    [parameter(Mandatory = $true)]
    [string] $Path,
    [parameter(Mandatory = $true)]
    [string] $Name
```

```

    )
$System = [Environment]::GetFolderPath("System")
$script:hostsPath = ([System.IO.Path]::Combine($System,
"drivers\etc\"))
➡+"hosts"

function New-localWebsite([string] $sitePath, [string] $siteName)
{
    try
    {
        Import-Module WebAdministration
    }
    catch
    {
        Write-Host "IIS PowerShell module is not installed. Please
install it
➡first, by adding the feature"
    }
    Write-Host "AppPool is created with name: " $siteName
    New-WebAppPool -Name $siteName
    Set-ItemProperty IIS:\AppPools\$Name managedRuntimeVersion v4.0
    Write-Host
    if(-not (Test-Path $sitePath))
    {
        New-Item -ItemType Directory $sitePath
    }
    $header = "www."+$siteName+".local"
    $value = "127.0.0.1 " + $header
    New-Website -ApplicationPool $siteName -Name $siteName -Port 80
    ➡-PhysicalPath $sitePath -HostHeader ($header)
    Start-Website -Name $siteName
    if(-not (HostsFileContainsEntry($header)))
    {
        AddEntryToHosts -hostEntry $value
    }
}

function AddEntryToHosts([string] $hostEntry)
{
    try
    {
        $writer = New-Object System.IO.StreamWriter($hostsPath, $true)
        $writer.Write([Environment]::NewLine)
        $writer.Write($hostEntry)
        $writer.Dispose()
    }
    catch [System.Exception]
    {
        Write-Error "An Error occured while writing the hosts file"
    }
}

```

```

}
function HostsFileContainsEntry([string] $entry)
{
    try
    {
        $reader = New-Object System.IO.StreamReader($hostsPath +
"hosts")
        while(-not($reader.EndOfStream))
        {
            $line = $reader.Readline()
            if($line.Contains($entry))
            {
                return $true
            }
        }
        return $false
    }
    catch [System.Exception]
    {
        Write-Error "An Error occurred while reading the host file"
    }
}

```

第一部分是一个参数块，你已经在第21章中进行了对应的学习。

```

param(
    [parameter(Mandatory = $true)]
    [string] $Path,
    [parameter(Mandatory = $true)]
    [string] $Name
)

```

该参数块看起来有点不同，它定义了一个**-Path**和一个**-Name**参数，并且这两个参数均为强制性参数。公平的是，当你运行该命令时，你需要这两个信息。

下一组的命令行看起来更加神秘。

```

$System = [Environment]::GetFolderPath("System")
$script:hostsPath = ([System.IO.Path]::Combine($System,
"drivers\etc\"))

```

```
➡+"hosts"
```

它们看起来并不像在做任何危险的事——类似**GetFolderPath**语句并不会导致任何报警。要想知道它们到底实现了什么功能，那么就需要将它们放到**Shell**中去执行。

```
PS C:\> $system = [Environment]::GetFolderPath('System')
PS C:\> $system
C:\Windows\system32
PS C:\> $script:hostsPath = ([System.IO.Path]::Combine
($system,"drivers\etc\"))+"hosts"
PS C:\> $hostsPath
C:\Windows\system32\drivers\etc\hosts
PS C:\>
```

\$script:hostsPath代码创建了一个新的变量。这样除了**\$system**变量之外，又有了一个新的变量。这几行命令定义了一个文件夹路径以及文件路径。请记住这几个变量的值，这样在学习该脚本过程中可以随时参照。

该脚本的后面包含了3个函数：**New-LocalWebsite**，**AddEntryToHosts**和**HostsFile ContainsEntry**。一个函数类似于包含在一个脚本中的某部分脚本：每个函数都代表着可以被单独调用的已打包的脚本块。你可以看到，每个函数都会定义一个或多个输入参数，尽管在上面的**Param()**块中并未看到。相反，它们采用了一种仅在函数中才合法的参数定义方法：在函数名称后面的括号中将参数罗列出来（和**Parameter()**块一样）。其实，这也可算作一种快捷方式。

如果查看该脚本，你不会看到这些函数被脚本本身调用，因此如果照搬这些脚本，那么脚本根本无法运行。但是在函数**New-LocateWebSite**中，你可以看到用了函数**HostsFileContainsEntry**。

```
if(-not (HostsFileContainsEntry($header)))
{
    AddEntryToHosts -hostEntry $value
}
```


同时，你也可以看到，函数AddEntryToHoses被该代码调用。该函数被嵌套在IF语句中。你可以在PowerShell中执行Help *IF*来获取更多的帮助信息。

```
PS C:\> help *IF*

Name                           Category  Module
-----
diff                            Alias
New-ModuleManifest             Cmdlet    Microsoft.PowerShell.Core
Test-ModuleManifest            Cmdlet    Microsoft.PowerShell.Core
Get-AppxPackageManifest        Function  Appx
Get-PfxCertificate             Cmdlet    Microsoft.PowerShell.S...
Export-Certificate             Cmdlet    PKI
Export-PfxCertificate          Cmdlet    PKI
Get-Certificate                Cmdlet    PKI
Get-CertificateNotificationTask Cmdlet    PKI
Import-Certificate             Cmdlet    PKI
Import-PfxCertificate          Cmdlet    PKI
New-CertificateNotificationTask Cmdlet    PKI
New-SelfSignedCertificate       Cmdlet    PKI
Remove-CertificateNotification... Cmdlet    PKI
Switch-Certificate             Cmdlet    PKI
Test-Certificate              Cmdlet    PKI
about_If                      HelpFile
```

HelpFile通常罗列在最后，比如这里的About-If。通过阅读该命令对应的结果集，你就可以看到IF语句的工作原理。在上面示例的上下文中，该语句会检查函数HostsFileContainsEntry返回的值是True还是False；如果返回False，就会调用函数AddEntryToHosts。该语句暗示New-LocalWebSite函数才是脚本中“最主要”的函数，或者称之为期望被运行并触发某些变更的函数。HostsFileContainsEntry 和 AddEntryToHosts函数看起来就像是函数New-LocalWebSite的功能函数——在需要时才会被调用。所以，此时我们需要关注New-LocalWebSite函数。

```
function New-localWebsite([string] $sitePath, [string] $siteName)
{
    try
    {
        Import-Module WebAdministration
```

```

}
catch
{
    Write-Host "IIS PowerShell module is not installed. Please
install it
➔first, by adding the feature"
}
Write-Host "AppPool is created with name: " $siteName
New-WebAppPool -Name $siteName
Set-ItemProperty IIS:\AppPools\$Name managedRuntimeVersion v4.0
Write-Host
if(-not (Test-Path $sitePath))
{
    New-Item -ItemType Directory $sitePath
}
$header = "www."+$siteName+".local"
$value = "127.0.0.1 " + $header
New-Website -ApplicationPool $siteName -Name $siteName -Port 80
➔-PhysicalPath $sitePath -HostHeader ($header)
Start-Website -Name $siteName
if(-not (HostsFileContainsEntry($header)))
{
    AddEntryToHosts -hostEntry $value
}
}

```

你可能不太理解Try块。快速查找对应的帮助文档（**Help *Try***）会显示**About_Try_Catch_Finally**帮助文档，其中阐述到：Try部分中的任何命令都有可能产生一个错误信息。如果确实产生了错误信息，那么就会执行Catch部分的命令。所以上面的命令可以解释为：该函数会尝试载入**WebAdministration**模块，如果载入失败，那么会显示一个错误信息。坦白讲，我们认为在发生错误时，应该完全退出该函数，但是在这里并非如此。所以当**WebAdministration**模块未成功载入时，你可以想象，这里会看到更多的错误信息。所以在执行该脚本之前，你必须保证**WebAdministration**模块可用！

Write-Host块主要用作帮助追踪脚本运行进度。下一个命令是**New-WebAppPool**。查看帮助文档，发现该命令包含在**WebAdministration**模块中，该命令的帮助文档阐述了其作用。接下来，**Set-ItemProperty**命令看起来像是对刚建立的AppPool对象设置某些选项。

看起来这里简单的**Write-Host**命令，仅是为了在屏幕上放置一个空行。确实如此。如果你查看**Test-Path**，你会发现它会检查一个给定的路径是否存在，在这个脚本中是指一个文件夹。如果不存在，那么脚本就会使用**New-Item**命令创建该文件夹。

变量**\$Header**在创建后被用作将**\$SiteName**转化为一个类似“**www.sitename.local**”的网址，同时**\$Value**变量用作添加一个IP地址。之后**New-WebSite**命令会在使用多个参数后被执行——你可以通过阅读该命令对应的帮助文档来查看各个参数的作用。

最后执行**Start-WebSite**命令。在帮助文档中有说明，该命令会启动对应的网站使其运行。此时就会调用**HostsFileContainsEntry**和**AddEntryToHosts**命令。它们会确保**\$Value**变量中的值对应的站点信息会以（IP地址-名称）格式被添加到本地**Hosts**文件中。

26.2 逐行检查

在前面的小节中，我们采用的是逐行分析该脚本，这也是我建议你们采用的方式。当你逐行查阅每一行时：

- 识别其中的变量，并找出其对应的值，之后将它们写在一张纸上。因为大部分情况下，变量都会被传递给某些命令，所以记下每个变量可能的值会帮助你预测每个命令的作用。
- 当你遇到一些新的命令时，请阅读对应的帮助文档，这样可以理解这些命令的功能。针对**Get**-类型的命令，尝试运行它们——将脚本中变量的值传递给命令的参数——来查看这些命令的输出结果。
- 当你遇到不熟悉的部分时，比如**[Environment]**，请考虑在虚拟机中执行简短的代码片段来查看该片段的功能（使用虚拟机有助于保护你的生产环境）。可以通过在帮助文档中搜寻（使用通配符）这些关键字来查阅更多的信息。

最重要的是，请不要跳过脚本中的任意一行。请不要抱有这种想法：“好吧，我不知道这一行命令的功能是什么，那么我就可以跳过它，继续看后面的命令。”请一定先停下来，找出每一行命令的作用或者你认为它们可以实现的功能。这样才能保证你知道需要修改哪些部分的脚本来满足特定的需求。

26.3 动手实验

注意： 对于本次动手实验来说，你需要运行PowerShell v3或更新版本PowerShell的计算机。

代码清单26.2呈现了一个完整的脚本。看看你是否能明白该脚本所实现的功能，以及实现的原理。你是否能找到该脚本中可能会出现的错误？需要如何修改该脚本才能使得可以在你的环境中运行？

请注意，你应该照搬该脚本，但是如果在你的系统中无法执行，你是否能够跟踪到问题所在？请记住，你应该见过该脚本里面的大部分命令，如果遇到没见过的命令，请查看PowerShell的帮助文档。帮助文档中的示例部分包含本脚本中用到的所有技术。

代码清单 26.2 Get-LastOn.ps1

```
function get-LastOn {
<#
.DESCRIPTION
Tell me the most recent event log entries for logon or logoff.
.BUGS
Blank 'computer' column

.EXAMPLE
get-LastOn -computername server1 | Sort-Object time -Descending |
Sort-Object id -unique | format-table -AutoSize -Wrap
ID          Domain          Computer      Time
--          -
LOCAL SERVICE    NT AUTHORITY      4/3/2012 11:16:39 AM
NETWORK SERVICE  NT AUTHORITY      4/3/2012 11:16:39 AM
SYSTEM           NT AUTHORITY      4/3/2012 11:16:02 AM

Sorting -unique will ensure only one line per user ID, the most
recent.
Needs more testing

.EXAMPLE
PS C:\Users\administrator> get-LastOn -computername server1 -
newest 10000
-maxIDs 10000 | Sort-Object time -Descending |
Sort-Object id -unique | format-table -AutoSize -Wrap
```

ID	Domain	Computer	Time
--	-----	-----	----
Administrator	USS		4/11/2012 10:44:57 PM
ANONYMOUS LOGON	NT AUTHORITY		4/3/2012 8:19:07 AM
LOCAL SERVICE	NT AUTHORITY		10/19/2011 10:17:22 AM
NETWORK SERVICE	NT AUTHORITY		4/4/2012 8:24:09 AM
Student	WIN7		4/11/2012 4:16:55 PM
SYSTEM	NT AUTHORITY		10/18/2011 7:53:56 PM
USSDC\$	USS		4/11/2012 9:38:05 AM
WIN7\$	USS		10/19/2011 3:25:30 AM

PS C:\Users\administrator>

.EXAMPLE

get-LastOn -newest 1000 -maxIDs 20

Only examines the last 1000 lines of the event log

.EXAMPLE

get-LastOn -computername server1 | Sort-Object time -Descending |
Sort-Object id -unique | format-table -AutoSize -Wrap
#>

param (

```

    [string]$ComputerName = 'localhost',
    [int]$Newest = 5000,
    [int]$maxIDs = 5,
    [int]$logonEventNum = 4624,
    [int]$logoffEventNum = 4647

```

)

\$eventsAndIDs = Get-EventLog -LogName security -Newest
\$Newest |

Where-Object {\$_.instanceid -eq \$logonEventNum -or
➔ \$_.instanceid -eq \$logoffEventNum} |

Select-Object -Last \$maxIDs

➔ -Property TimeGenerated,Message,ComputerName

foreach (\$event in \$eventsAndIDs) {

```

    $id = ($event |
    parseEventLogMessage |
    where-Object {$_.fieldName -eq "Account Name"} |
    Select-Object -last 1).fieldValue

```

```

    $domain = ($event |
    parseEventLogMessage |
    where-Object {$_.fieldName -eq "Account Domain"} |
    Select-Object -last 1).fieldValue

```

\$props = @{'Time'=\$event.TimeGenerated;

```

        'Computer'=$ComputerName;
        'ID'=$id
        'Domain'=$domain}

        $output_obj = New-Object -TypeName PSObject -Property
$props
        write-output $output_obj
    }
}

function parseEventLogMessage()
{
    [CmdletBinding()]
    param (
        [parameter(ValueFromPipeline=$True,Mandatory=$True)]
        [string]$Message
    )

    $eachLineArray = $Message -split "`n"

    foreach ($oneLine in $eachLineArray) {
        write-verbose "line:_$oneLine_"
        $fieldName,$fieldValue = $oneLine -split ":", 2
        try {
            $fieldName = $fieldName.trim()
            $fieldValue = $fieldValue.trim()
        }
        catch {
            $fieldName = ""
        }

        if ($fieldName -ne "" -and $fieldValue -ne "" )
        {
            $props = @{'fieldName'="$fieldName";
                        'fieldValue'=$fieldValue}

            $output_obj = New-Object -TypeName PSObject -
Property $props
            Write-Output $output_obj
        }
    }
}
Get-LastOn

```

第27章 学无止境

你基本上完成了对本书的学习，但是请不要停止对PowerShell的进一步学习。其实，在PowerShell中还有更多值得学习的东西。基于我们在本书中学到的知识，你在后面可以进行大量的自学。本章是一个小章节，但是本章会给你指出一些正确的学习方向。

27.1 进一步学习的思想

本书主要关注于希望成为高效的PowerShell用户所需掌握的技能与技术。换句话说，你应该能使用PowerShell中上千可用的命令来完成一些任务，而不论你的需求是关于Windows、Exchange、SharePoint还是其他产品。

下一步需要完成的是将多个命令结合在一起构成一个包含多个步骤的自动化流程，例如针对第三方人群建立一个已打包的可随时使用的工具。我们称之为工具制作（ToolMaking）。如果要详细描述该过程，可能需要一整本书的篇幅来介绍。但是也可以通过本书中所学的知识，编写一些参数化的脚本。在这些脚本中可以包含你所需的各种命令，之后借助该参数化脚本来完成某项任务——其实，这也就是工具制作的初级阶段。

如果需要完成工具制作，需要包含哪些东西呢？

- PowerShell的简化编程语言；
- 作用域；
- 功能，以及将多个工具整合到单个脚本文件的能力；
- 错误处理；
- 帮助文档的编写；
- 调试；
- 自定义显示格式；
- 自定义类型扩展；
- 脚本与清单模块；
- 使用数据库；
- 工作流；

- 管道排错；
- 复杂的对象层次结构；
- 全局对象与本地对象；
- 可视化的PowerShell工具；
- 代理功能；
- 受限的远程处理与委托管理；
- .Net的使用。

其实还有更多需要用到的东西。如果你有足够的兴趣并且掌握适当的技能，你甚至可以成为PowerShell的第三方观众的一部分——也就是软件开发者。有一整套围绕开发PowerShell的工具以及在开发过程中使用PowerShell的工艺和技术。这是多么伟大的一个产品啊！

27.2 既然已经阅读了本书，那么我要从哪里开始呢

现在最应该做的就是选择一个任务。选取真实环境中一些重复性的工作，然后利用PowerShell工具使得可以自动化完成该项工作。你肯定会碰到某些不知道该如何做的事情，那么这就是开始学习的最好的切入点。

下面是我们看到的其他管理员遇到的一些事情。

- 编写一段脚本修改某服务的登录账号的密码，并且将该脚本发送到运行该服务的多台计算机上（可以使用单行命令实现）。
- 编写一段脚本，用来实现新用户配置的自动化处理，包含新建用户账号、用户邮箱以及根目录等。通过PowerShell来配置NTFS权限会稍微麻烦点，所以请考虑使用基于PowerShell脚本开发的Caccls.exe或者Xcaccls.exe，而不要使用PowerShell的Get-ACL以及Set-ACL命令（这两个命令使用起来都比较复杂）。
- 编写管理Exchange邮箱的脚本——如获取占据空间最多的邮箱的报表或者针对邮箱大小创建一个报表。
- 通过包含在Windows Server 2008 R2以及之后操作系统中的WebAdministration模块实现IIS中自动化发布新站点（如果是Windows Server 2008中采用IIS7，也可实现）。

记住，最重要的一点是“不要考虑太多”。Don曾经遇到一个管理员，该管理员花费好几个星期编写了一段PowerShell脚本来实现强大的文件拷贝功能，这样他就可以通过Web Server进行发布。Don问道：“为什么不直接使用XCopy或者RoboCopy呢？”该管理员盯着Don看了一会儿，然后笑了。其实，该管理员陷入了一个误区：“仅使用PowerShell来实现”，他忘记了“PowerShell可以直接调用那些已存在的强大的组件”。

27.3 你会喜欢的其他资源

我们花费了大量的时间去使用PowerShell，编写PowerShell方面的书籍以及进行PowerShell相关的教学工作。不信可以询问我们的家人——有时甚至我们只有在吃饭的时候才不谈论PowerShell。这就意味着，我们积累了很多的在线资源——包含日常工作中使用的，以及给学生建议的。希望这些资源也能给你提供一个很好的学习出发点。

- **MoreLunches.com**——如果你还没将该网站加入书签中，那么该地址将是你的第一站。在该网站上，你会发现针对该书的免费福利以及配套内容，其中包括动手实验环节的答案、视频演示、免费文章以及额外的推荐资源。你也可以下载本书中那些很长的代码清单，这样就不用手动输入这些命令。请将该网站加入书签页中，然后定期访问该网站，以便对本书中所学的知识加深印象。
- <http://PowerShell.org> ——Don与很多专家一起在该社区站点上发表了博客文章。
- <http://jdhitsolutions.com/blog> ——这是Jeff的发布通用脚本以及PowerShell相关文章的博客站点。
- <http://mcpmag.com/Articles/List/Prof-PowerShell.aspx> ——这是Jeff为MCP Mag.Com站点撰写的“Prof.PowerShell”周刊，里面全是简短的一些教程以及技巧。
- <http://PowerShell.org> ——在该社区上包含一个公开的PowerShell Q&A论坛，我们会直接在该论坛上回答大家的PowerShell相关的问题。

很多学生经常都在问：是否还有其他一些推荐的书籍？在我们的桌上仅摆放了少量书籍，这些书籍名称都存在于

<http://PowerShellBooks.org/wp/books> 网站列表中。当有新出版的书籍时，该列表会进行更新。其中的两本Learn PowerShell Toolmaking in a

Month of Lunches以及**PowerShell In Depth**（均可在**Manning**上购买）是由我们编写或者合著的。所以如果你喜欢这两本书，那么这两本书会对你有很大的帮助。

最后，如果你喜欢**PowerShell**相关的未删节视频类型的培训，那么请访问<http://CBTNuggets.com> 网站。在该网站上，**Don**和其他**PowerShell**专家提供了一些未删节的高清视频。请记住，**MoreLunches.com**网站也提供了本书中每章节对应的配套视频，并且这些视频是免费的。

第28章 PowerShell备忘清单

现在是时候将遇到的一些小问题进行整理了。当你遇到什么问题时，请记住首先翻到本章进行查找。

28.1 标点符号

毫无疑问，PowerShell命令中包含了大量的标点符号，并且大部分的标点符号在帮助文档和PowerShell中具有不同的含义。下面是这些标点符号在PowerShell中的含义。

- ```（重音符）——重音符是PowerShell中的转义字符。它会移除紧跟在重音符后面的特定字符串的作用。例如，通常情况下，空格符是一个分隔符，这也就是在PowerShell中`cd C:\Program Files`会执行失败的原因。将该空格符转义，`cd Program`Files`，会将该空格的作用去除，仅将该符号作为文字中的一部分。这样这个命令就可以正常执行了。
- `~`（波浪符）——当将`~`作为路径的一部分时，该字符表示当前用户的根目录，也就是在系统变量UserProfile中定义的值。
- `()`（括号）——有两种使用场景：

-和在数学中一样，括号定义了执行的顺序。PowerShell会优先执行括号中的命令。如果存在多重括号，则会从最里层括号向外执行。通过这种方式，可以很轻易实现：先执行一个命令，之后将该命令的输出结果传递给另外一个命令的某个参数，比如`Get-Service - ComputerName (Get-Content C:\ComputerNames.txt)`。

-括号也可以被用作包含一个方法的参数。即使该方法不要求使用任何参数，也必须带有括号，比如`Change-Start-Mode('Automatic')`以及`Delete()`。

- `[]`（方括号）——在PowerShell中有两种使用方式：

-需要访问一个数组或者集合中某个单独的对象时，可以使用方括号来指定对应的索引号：`$Services[2]`表示从`$Services`中获取第三个对

象（请记住索引编号是从0开始计数的）。

-当需要将某个数据转化为特定的类型时，需要将类型包含在方括号中。例如，`$My Result/3 as [INT]`会将除法运算的结果转化为整数；再比如，命令`[XML]$Data=Get-Content Data.XML`会读取Data.XML中的内容，并且尝试将该内容解析为合法的XML文件。

- {}（花括号）——有三种用途：

-花括号可用作包含可执行代码或者命令块，我们称之为脚本段（Script Blocks）。该脚本段经常被作为值传递给那些可接受脚本段或者筛选块的参数：`Get-Service | Where-Object{$_ .Status -eq 'Running'}`。

-花括号可用作包含构成哈希表的键-值对。左大括号前面总是一个“@”符号。在下面的示例中，我们使用花括号来包含哈希表的键-值对（在示例中，有两组键-值）。第二个花括号包含一段表达式的脚本段，该脚本段作为第二个键的值：`$HashTable= @{l='Label';e={expression}}`。

-当变量的名称中包含空格或者其他非法字符时，必须使用花括号来包含这部分信息：`${My Variable}`。

- ' '（单引号）——单引号可用作包含字符串（String）。PowerShell并不会对包含在单引号中的字符串查找转义字符或者变量。
- " "（双引号）——双引号也可用作包含字符串，但与单引号不同的是，PowerShell会针对双引号中的字符串数据进行查找转义字符以及\$字符。其中会进行针对转义字符的处理，同时\$符号后面带有的字符（到下一个空格为止）会被识别为一个变量名字，并且其值会被替换掉。例如，如果变量\$One的值为“World”，同时定义\$Two="Hello \$One `n"，那么\$Two的值就会是“Hello World”之后再加一个回车（`n代表一个回车键）。
- \$（美元符号）——该符号告诉PowerShell \$后面的字符（截止到下一个空格处）为一个变量的名称。但是当在使用管理变量的Cmdlet时，可能容易造成误解。假如\$One变量的值为Two，然后执行New-Variable -Name \$One -Value 'Hello'命令，会创建一个名为Two的变量，并且其值为“Hello”——有些人很奇怪，为什么变

量的名字会是Two。这是因为\$符号告诉PowerShell使用\$One的值来作为新变量的名称。相对应地，New-Variable -Name One - Value 'Hello'，该命令会创建一个名为One的变量。

- %（百分号）——百分号是ForEach-Object Cmdlet的别名，同时它也是模运算符，返回除法运算后的余数。
- ?（问号）——问号是Where-Object Cmdlet的别名。
- >（右尖括号）——该符号类似Out-File Cmdlet的一个别名。但是严格来讲，它并不是一个真正的别名，但是却提供了Cmd.exe类型的文件重定向功能：Dir>Files.Txt。
- + - * / %（数学运算符）——这些运算符是作为标准算术运算符使用。请注意，+也可以用作字符串连接使用。
- -（破折号或者叫连字符）——可以用作连接参数名称或者其他运算符，如-Co
- mputerName或者-Eq。同时破折号也可以用作分离Cmdlet名称中的动词与名词，比如Get-Content。另外，破折号也作为算术中的减法运算符使用。
- @（at符号）——在PowerShell中有四个用途：

-可用在左花括号前面（请参阅上面的介绍花括号部分）。

-当用在括号之前时，它会包含组成数组的一串以逗号分隔的值：\$Array= @(1,2,3,4)。其中的@字符与括号是可选的，因为PowerShell默认会将以逗号分隔的列表识别为数组。

-可以指一个Here-String。Here-String是指包含在单引号或者双引号中的字符串。一个Here-String以“@”字符作为开始和结束的标志，结束的“@”必须位于另起一行的起始位置。如果想获取更多的信息或者示例，请执行Help About_Quoting_Rules。另外需要说明的是，Here-String也可通过单引号进行定义。

-@也是PowerShell中的传递符（Splat Operator）。如果构建了一个哈希表，在哈希表中，键名称能匹配参数名称，并且键的值为参数的值，那么你就可以将该哈希表传递给一个Cmdlet。Don曾经为TechNet Magazine写过一篇关于传递（Splating）的文章（<https://technet.microsoft.com/en-us/magazine/gg675931.aspx>）。

- &（与符号）——这是PowerShell中的一个调用运算符，使得PowerShell可以将某些字符识别为命令，并运行这些命令。例

如，`$a="Dir"`命令将“Dir”赋给了变量`$a`，然后`&$a`就会执行Dir命令。

- `;`（分号）——分号一般用作分隔PowerShell中同一行的两个命令：`Dir;Get-Process`。这个命令会先执行Dir命令，之后执行Get-Process命令。它们的执行结果会发送给一个管道，但是Dir命令的执行结果并不会通过管道发送给Get-Process命令。
- `#`（井号）——该符号为注释符号。跟在#之后的文字，到下一个回车之前，均会被PowerShell忽略掉。尖括号`<>`可以被用作定义一个注释块的标签，“`<#`”作为起始，“`#>`”作为结束。包含在该注释块中的所有命令均会被PowerShell忽略掉。
- `=`（等号）——等号是PowerShell中的赋值运算符，用来向一个变量进行赋值：`$One=1`。但是它不能用作数量比较，相反需要使用`-Eq`。另外需要记住，该运算符可以与数学运算符结合使用：`$Var+=5`。该命令会对`$Var`变量的值增加5。
- `|`（管道符）——管道符主要用作将一个Cmdlet的输出结果传递给另外一个Cmdlet。第二个Cmdlet（接收输出结果的Cmdlet）采用管道参数绑定方法来确定哪个参数或者哪些参数来负责接收传入的管道对象。第9章中对该过程进行了讲解。
- `\`或者`/`（反斜杠或斜杠）——斜杠可以作为数学表示中的除法运算符；反斜杠和斜杠也可以作为文件路径中的分隔符：`C:\Windows`和`C:/Windows`路径一致。反斜杠在WMI筛选场景以及正则表达式中也可作为转义字符。
- `.`（句号）——句号有三种用途：

-句号可以被用作表示希望访问某个成员，比如一个属性或方法；再或者一个对象：`$_Status`表示访问`$_`占位符中对象的Status属性。

-它可以通过“`.`”引用源码来执行一段脚本，意味着该脚本运行在当前作用域下，并且该脚本定义的任何对象在脚本运行完毕之后均存在，比如`C:\myscript.ps1`。

-两个“`.`”（`..`）会形成一个范围运算符，该运算符在本章后面会讲到。你也会发现，“`..`”也可用作表示文件系统中的当前路径的父文件夹，比如`..\`。

- `,`（逗号）——当用在引号外面时，逗号可以用作分隔数组或者列表中的项：`"One",2,"Three",4`。另外，它也可用作将多个静态值传

递给可接收这些值的参数：`Get-Process -ComputerName Server1,Server2,Server3`。

- `:`（冒号）——冒号（严格来说应该是两个冒号）可用作访问类的静态成员。这里采用了`.Net Framework`编程语言的概念，比如`[DateTime]::Now`（其实也可以使用`Get-Date`来获取相同的结果）。
- `!`（感叹号）——是“非”(Not)布尔运算符的别名。

我们认为，在美国键盘格式中没有被`PowerShell`使用到的应该是脱字符“`^`”，毕竟该符号常用于正则表达式运算。

28.2 帮助文档

帮助文档中的标点符号与`PowerShell`中相比，具有略微不同的含义。

- `[]`——大括号，用作表达包含在大括号中的文本为可选项。比如包含在其中的所有命令（`[-Name <String>]`）；或者当参数是位置参数时，参数名称可选（`[-Name] <String>`）。也可用作表达下面两个含义：参数是可选项，并且如果指定了该参数，那么该参数可作为位置参数使用（`[[Name] <String>]`）。如果你觉得有任何问题，请在命令中指定参数名称，因为这样始终是符合语法规范的。
- `[]`——相邻的大括号表示一个参数可接受多个值（`<String>[]`，而非`<String>`）。
- `<>`——尖括号可用来包含数据类型，表示值的类型或者参数匹配的对象：`<String>`，`<int>`，`<Process>`等。

请一定要养成阅读完整帮助文档的好习惯（对`Help`命令添加`-Full`参数），因为通过该命令会提供尽可能详细的信息，大多数情况下会包含示例。

28.3 运算符

`PowerShell`不会使用其他编程语言使用的常规比较运算符。相反，它使用下列运算符。

- **-eq**——等于（**-ceq**用作字符串比较，包括大小写是否一致）。
- **-ne**——不等于（**-cne**用作字符串比较，包括大小写是否一致）。
- **-ge**——大于或等于（**-cge**用作字符串比较，包括大小写是否一致）。
- **-le**——小于或等于（**-cle**用作字符串比较，包括大小写是否一致）。
- **-gt**——大于（**-cgt**用作字符串比较，包括大小写是否一致）。
- **-lt**——小于（**-clt**用作字符串比较，包括大小写是否一致）。
- **-contains**——若数据集包含特定对象，则返回真（**True**）。（**\$Collection -Contains \$Object**。）**-nocontains**表示相反含义。
- **-in**——若特定对象包含在数据集中，则返回真（**True**）。（**\$Object -in \$Collection**。）**-notin**表示相反含义。

逻辑运算符可用作组合运算：

- **-not**——将真假值取反（**!**是该运算符的别名）。
- **-and**——如果整个表达式要为真，则所有子表达式均需要为真。
- **-or**——如果整个表达式要为真，则其中一个子表达式需要为真。

另外，还存在执行特定操作的运算符：

- **-Join**——将一个数组的元素连接为分隔的字符串。
- **-Split**——将一个分隔的字符串分离为一个数组。
- **-Replace**——将一个字符串中特定字符（串）替换为另外的字符（串）。
- **-Is**——若一个对象为指定类型，返回为真（**True**）。（**\$ID -Is [INT]**）
- **-As**——将对象转化为特定类型（**\$ID -As [INT]**）
- **..**——一个范围运算符，**1..10**会返回1到10的十个对象。
- **-F**——格式化运算符，会使用后面提供的值替换对应的占位符。（**"{0},{1}" -F "Hello","World"**）

28.4 自定义属性与列的语法

在多个章节中，我们曾经演示如何使用**Select-Object**来定义自定义属性，或者分别使用**Format-Table**以及**Format-List**来自定义列或列表条目。下面是对应的哈希表语法。

可以通过该语句得到每一个自定义属性或者列：

```
@{Label='Column_or_Property_Name';Expression={Value_Expression}}
```

这里的两个键“Label”和“Expression”，可以分别缩写为“l”和“e”（请注意，这里是小写的字母l，不是数字1）。当然，你也可以使用n作为键的名称。

```
@{n='Column_or_Property_Name';e={Value_Expression}}
```

在表达式中，可以使用\$_占位符关联到当前对象（比如当前表中的行或者期望添加自定义属性的对象）。

```
@{n='ComputerName';e={$_.Name}}
```

Select-Object和Format- 的Cmdlet均会查找n（或者name或者label或者l）键和e键；Format- Cmdlet也支持Width和Align（仅支持Format-Table）和FormatString操作。请阅读Format-Table命令的帮助文档，获取对应的示例。

28.5 管道参数输入

在第9章中我们看到，在PowerShell中有两种方式进行参数绑定：ByValue和ByPropertyName。优先使用ByValue方法，仅当ByValue方法无法执行时才会尝试使用ByPropertyName方法。

对ByValue方法而言，PowerShell会查看放入管道中对象的类型。当然，你也可以通过gm命令自行查看该对象的类型名称。之后PowerShell会检查该Cmdlet中是否有参数可以接收传入的对象类型，并且检查是否有参数可以使用ByValue方法来接收管道输入。对一个Cmdlet而言，如果采用这种方式，则不可能有两个参数绑定到相同的数据类型。换句话说，你无法看到一个Cmdlet中有两个参数均满足如

下两个条件：均可接收<String>类型的输入，均可使用ByValue方法实现参数绑定。

如果无法使用ByValue方法，那么PowerShell就会尝试使用ByPropertyName方法。在该方法中，PowerShell仅简单查看放入管道中对象的属性，之后尝试找到某个可接收通过ByPropertyName方法传入对象的参数，并且要求该参数的名称与属性名称一致。例如，如果放入管道中的对象包含Name、Status和ID属性，PowerShell会查看Cmdlet中是否有参数名为Name、Status和ID。同时要求这些参数被标记为“可接收ByPropertyName管道输入”。至于如何查看是否满足条件，请阅读对应的详细帮助文档（记住，在使用Help命令时加上-Full参数）。

让我们看看PowerShell如何实现这些功能。比如本例，假如有一个命令为Get-Service | Stop-Service或者是Get-Service | Stop-Process，将其中第一个Cmdlet称为第一个命令，类似地，第二个Cmdlet称为第二个命令。PowerShell采用下面的步骤进行工作。

（1）第一个命令产生的对象类型是什么？你可以将该Cmdlet输出结果通过管道传递给Get-Member来自行查看该信息。对那些名称由多部分字符组成的类型而言，仅需记住最后一位（比如类型名称为System.Diagnostics.Process，仅需记住最后一位的Process即可）。

（2）第二个命令中是否有参数可以接收第一个命令产生的对象类型（通过查看第二个命令对应的详细帮助文档进行确定：Help <Cmdlet> -Full）？如果存在，那么再检查该参数是否可以接收通过ByValue方式传入的管道对象。每个参数对应的帮助文档中的详细说明中均包含该信息。

（3）如果步骤（2）的答案是Yes，那么第一个命令产生的完整对象就会被关联到步骤（2）中满足条件的参数。此时，所有步骤就结束了——不需要再到步骤（4）。但是如果步骤（2）的答案是“否”，那么就需要继续步骤（4）。

（4）此时需要检查第一个命令产生的对象。查看产生的对象包含什么属性。再次说明，你可以通过将第一个命令产生的对象通过管道传递给Get-Member来查看该信息。

(5) 此时检查第二个命令的参数（此时需要重新查看详细帮助文档）。是否有参数的名称与步骤（4）中找到的属性名称一致（条件a），并且该参数是否能接收通过ByPropertyName方式传入的对象（条件b）？

(6) 如果有任一参数满足步骤（5）中的a和b条件，那么第一个命令产生对象的属性就会关联到对应的第二个命令的同名参数，第二个命令就会运行。如果第一个命令产生对象的属性名称与第二个命令中可接收ByPropertyName方式传入对象的参数名称不一致，那么第二个命令也会运行，但是此时第二个命令并没有管道输入。

另外需要注意的是，你可以针对任意命令手动输入参数以及其值。但是此时，将会导致参数无法接收管道输入对象，即使正常情况下可以使用某种管道输入方法（不管是ByValue还是ByPropertyName）。

28.6 何时使用\$_

这或许是PowerShell中最让人费解的问题之一：什么时候才能使用\$_占位符？

当PowerShell显式查找\$_，并且准备使用其他数据填充该占位符时，可以使用\$_占位符。一般来讲，这只会发生在处理管道输入的本段中——在这种情况下，\$_占位符一次只能包含一个管道输入对象。在下面几个不同的地方会用到该占位符。

- 在Where-Object的筛选脚本段中：

```
Get-Service | 3 Where-Object { $_.Status -eq 'Running' }
```

- 在传递给ForEach-Object命令的脚本段中，比如下面命令中使用的-Process脚本段：

```
Get-WmiObject -class Win32_Service -filter "name='mssqlserver'" |  
ForEach-Object -process { $_.ChangeStartMode('Automatic') }
```

-
- 针对过滤功能和高级功能的**Process**脚本段。我们编写的另外一本书中讨论到该部分知识——*Learn PowerShell Toolmaking in a Month of Lunches*。
 - 用来创建自定义属性或者表列的哈希表表达式中，请参考28.4小节查看更多细节，或者阅读第8、9、10章中更完整的讨论。

在上面所有场景中，**\$_**占位符仅会出现在脚本段的花括号中。那么这也是一个判断什么时候可以使用**\$_**占位符的比较好的规则。

附录 复习实验

当你完成这本书中指定的章节和实验后，可以继续完成本篇附录中提供的3个复习实验。对于你的学习过程来说，复习是一种很好的休息方式，同时可以巩固你已经学到的最为重要的要点。和往常一样，你可以从MoreLunches.com 网站上找到示例答案。通过找到这本书的封面图片，单击它，然后去下载区下载实验示例解决方案文件即可。

因为这些实验任务中的一部分实验说明命令较为复杂，所以我们将这些复杂的说明命令分解为独立的任务小节。同时为了帮助你完成实验，在每个实验开端，我们也提供了一个提示清单来提示你，包括你可能会需要的特定命令、帮助文件和语法。

实验回顾1：第1—6章

注意： 为了完成这些实验，你需要一台运行PowerShell v3或更新版本的PowerShell的计算机。在打算完成这些实验之前，你应该先完成这本书中的第1—6章的实验。

提示：

- Sort-Object
- Select-Object
- Import-Module
- Export-CSV
- Help
- Get-ChildItem (Dir)

任务1

运行一个命令，从而显示应用程序事件日志中最新的100个条目，不要使用Get-WinEvent。

任务2

写一个仅显示前五个最消耗虚拟内存（**VM**）进程的命令。

任务3

创建一个包含所有的服务**CSV**文件，只需要列出服务名称和状态。所有处于运行状态的服务处于停止状态的服务之前。

任务4

写一个命令行，将**BITS**服务的启动项类型变更为手动。

任务5

显示你计算机中所有文件名称为**Win.**的文件，以**C: **开始。注意：为了完成这个实验，你可能需要去实验和使用一些**Cmdlet**命令的新参数。

任务6

获取一个**C:\Program Files**的目录列表。包含所有的子文件夹，把这些目录列表放到位于**C:\Dir.txt**的文本文件内（记住去使用**the >redirector**, 或者 **Out-FileCmdlet**）。

任务7

获取最近**20**条安全事件日志的列表，将这些信息转化成**XML**格式。不要在硬盘上创建文件，而是把**XML**在控制台窗口直接显示出来。

注意： 该**XML**可以作为一个单独的原生对象显示，而不是以一个原始的**XML**数据。这没问题。那也是**PowerShell**展示**XML**的方式。如果你喜欢，你可以将**XML**对象通过管道传递给**Format-Custom**命令，从而查看**XML**展开为对象层级的形式。

任务8

获取一个服务列表，并将其导出到以**C:\services.csv**命名的**CSV**文件内。

任务9

获取一个服务列表，仅保留服务名称、显示名称和状态，然后将这些信息发送到一个HTML文件。在HTML文件中的服务信息表格之前显示“Installed Services”。

任务10

为Get-ChildItem创建一个新的别名D。仅将别名导出到一个文件里。关闭这个Shell，然后打开一个新的控制台窗口。把别名导入到新的Shell中。确认能够通过运行D并且获得一个目录列表。

任务11

显示系统中存在的事件日志列表。

任务12

运行一个命令来展示Shell所在的当前目录。

任务13

运行一个命令，展示最近你在Shell中运行过的命令。从中查找你在任务11中所运行的命令。将这两个命令通过管道传输符进行连接，重新运行任务11的命令。

换句话说，假如Get-Something 是一个获取历史命令的命令，5是任务11的命令ID号，并且Do-Something 是运行历史命令的命令，运行如下。

```
Get-Something -id 5 | Do-Something
```

当然，上面的命令并不是正确的命令，你需要找到正确的命令。

提示： 你所需寻找的两个命令有相同的名词。

任务14

运行一个命令，从而在需要时通过覆盖旧日志来修改安全事件日志。

任务15

通过使用New-Item Cmdlet来创建一个名称为C:\Review的新目录。这与运行Mkdir是不一样的；New-Item 命令需要知道你所想要创建的新项目是什么类型。通过命令读取帮助信息。

任务16

显示该注册码的内容：

```
HKCU:\Software\Microsoft\Windows\CurrentVersion\Explorer\User  
Shell Folders
```

注意： “User Shell Folders”与真正意义上的目录并不一样。如果你改变该“目录”，你将不能在目录清单中看到任何条目。User Shell Folders 是一个项目，其包含的是项目属性。有一个Cmdlet能展示属性项（尽管命令使用的是单数名词而不是复数）。

任务17

找出（但是请不要运行）命令能做如下事情的：

- 重启电脑；
- 关闭电脑；
- 从工作组或者域内移除一个电脑；
- 恢复一个电脑系统，并重建检查点。

任务18

你认为什么命令可以改变一个注册表值？提示：它是一个和你在任务16中发现的命令相同的名词。

实验回顾2：第1—14章

注意：为了完成这些实验，你需要一台运行PowerShell v3或更新版本的PowerShell的计算机。在打算完成这些实验之前，你应该先完成这本书中的第1—14章的实验。

提示：

- Format-Table
- Invoke-Command
- Get-Content(or Type)
- Parenthetical commands
- @{label='columnheader';expression={\$.property}}
- Get-WmiObject
- Where-Object
- -eq -ne -like -notlike

任务1

在一个表格中展示一个正在运行的进程的列表，其中只包含进程的名字和ID号。不要让这个表格在两列之间有大的空白区域。

任务2

运行如下命令：

```
Get-WmiObject -class Win32_UserAccount
```

现在再一次运行相同的命令，但是将内容格式化输出到一个有Domain和UserName列的表格中。UserName列应该显示用户的Name属性，如下：

```
Domain  UserName
=====  =====
COMPANY  DonJ
```

确保这个第二列标题叫UserName，而不是Name。

任务3

让两台电脑（也可以使用Localhost两次）运行如下命令：

```
Get-PSProvider
```

使用远程处理去做，确保输出包含计算机名称。

任务4

使用Notepad 创建一个名为C:\Computers.txt 的文件。在文件中写入如下内容：

```
Localhost  
Localhost
```

你应该确保上述两个名称各自独占一行——总共2行。保存文件并关闭记事本。然后写一个命令列出正在电脑上运行的服务名称写入到C:\Computer.txt。

任务5

查询Win32_LogicalDisk的所有实例。仅显示DriveType属性中包含3且有百分之五十以上的可用磁盘空间的实例。

提示：计算可用空间百分比，公式为 $\text{freespace}/\text{size} * 100$ 。

注意，Get-WmiObjectcannot 的过滤参数中无法包含数学表达式。

任务6

显示在root\CIMv2的命名空间下的所有的WMI类列表。

任务7

在列表中显示所有StartMode是Auto且State属性不是Running的Win32_Service的实例。

任务8

找到一个能发送Email信息的命令。这个命令的必要参数都是什么？

任务9

运行一个显示C:\下目录权限的命令。

任务10

运行一个可以显示所有C:\Users下子文件夹权限的目录，仅包含直接子文件夹，不需要去递归所有的文件和文件夹。你需要把一个命令的结果通过管道传输给另一个命令，即可实现。

任务11

找到一个可以使用其他凭据而不是当前登录用户的凭据启动记事本的命令。

任务12

运行一个命令，使Shell暂停或者闲置10秒。

任务13

你能找到帮助文件来解释Shell的各种运算符吗？

任务14

写一个信息类消息到应用事件日志。日志类别为1，原始数据为100000。

任务15

运行如下命令：

```
Get-WmiObject -Class Win32_Processor
```

了解该命令的默认输出结果。现在，修改这个命令，使得输出结果在表格里显示。表格内容应该包含每个处理器的核心数、制造商和名称，也包括一个列名为“MaxSpeed”的列，该列表示处理器的最大时钟频率。

任务16

运行如下命令：

```
Get-WmiObject -Class Win32_Process
```

了解这个命令的默认输出。如果希望的话，可以将该输出结果通过管道传递给Get-Member命令。现在，将该命令修改为仅显示在峰值情况下工作集超过5000的处理器。

实验回顾3：第1—19章

注意： 为了完成这些实验，你需要一台运行PowerShell v3或更新版本的PowerShell的计算机。在打算完成这些实验之前，你应该先完成这本书中的第1—19章的实验。

从回答下列问题开始：

1. 你会使用哪一个命令启动一个完全在你本地计算机运行的作业？
2. 你会使用哪一个命令启动一个作业的内容被远程计算机处理但由本地计算机调整的作业？
3. \${computer name}是一个合法的变量名称吗？
4. 你会如何展示由当前Shell定义的变量列表？
5. 哪一个命令可以被用来提示用户输入？

6. 哪一个命令可以被通常用于生成显示在屏幕上的输出结果，但也可以被重新转为多种其他输出格式？

现在完成以下三个任务：

任务1

创建一个处于运行状态的进程列表，该列表应该仅包含进程名称、ID、VM和PM。把这个列表放入一个名称为C:\Procs.html的HTML文件中。确保HTML文件有一个标题为“Current Processes”。在浏览器中显示文件，并把标题显示在浏览器窗口的标题栏中。

任务2

创建一个包含所有你的电脑上的服务的制表符定界文件，命名为C:\Services.tdf。“`t”(在双引号之间的反撇号t)是PowerShell为水平制表符使用的转义字符。文件中仅包含服务的名称、显示名称和状态。

任务3

重复任务1，将命令修改为在HTML文件中Vm列和PM列显示的值以MB为单位，而不是字节。计算兆字节的公式，以一个整体数字的数值显示，公式如下： $\$_{VM} / 1MB -as[int]$ for the VM property。

看完了

如果您对本书内容有疑问，可发邮件至contact@epubit.com.cn，会有编辑或作译者协助答疑。也可访问异步社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@epubit.com.cn。

在这里可以找到我们：

- 微博：@人邮异步社区
- QQ群：368449889

091507240605ToBeReplacedWithUserId